Centro Federal de Educação Tecnológica de Minas Gerais
Coordenação de Eletrônica
Curso Técnico de Eletrônica

UMA INTRODUÇÃO AO C2000 LAUNCHPAD

Gustavo Sathler Zappulla

Belo Horizonte

Centro Federal de Educação Tecnológica de Minas Gerais
Coordenação de Eletrônica
Curso Técnico de Eletrônica

UMA INTRODUÇÃO AO C2000 LAUNCHPAD

Relatório Técnico Final apresentado ao Centro Federal de Educação Tecnológica de Minas Gerais, como parte dos requisitos necessários à obtenção do grau de Técnico em Eletrônica.

Gustavo Sathler Zappulla

Coordenador de Curso(a): Ari Divino Soares, Esp.
Orientador(a): Prof. Rubens Marcos dos Santos Filho, Dr. E.E.

Belo Horizonte
2016

Aprovação

_____

Nome e assinatura do estagiário

_____

Nome e assinatura do supervisor da empresa

_____

Nome e assinatura do professor orientador

Belo Horizonte, ____ de _____ de 20____

Resumo

O kit de desenvolvimento *Piccolo Launchpad* é baseado no microcontrolador *TMS320F28027*, produzido pela *Texas Instruments*, e que pertence à família *Piccolo* da linha *C2000* para controle em tempo real.

Neste trabalho, desejamos fornecer um ponto de partida para o desenvolvimento de aplicações no Piccolo Launchpad. Ele será estruturado como um tutorial. Os primeiros tópicos apresentam as características do F28027, o ambiente de desenvolvimento *Code Composer Studio*, os modelos de programação e o passo-a-passo para a criação de novos projetos.

Em seguida, abordaremos os periféricos *ADC (Analog Digital Converter)* e *ePWM (Enhanced Pulse Width Modulator),* explicando seus princípios de funcionamento e dando exemplos de códigos comentados, de forma a facilitar o entendimento por parte do leitor.

SUMÁRIO

Aprovação 3
SUMÁRIO 5
1. INTRODUÇÃO 6
2. CARACTERÍSTICAS DO MICROCONTROLADOR 7
2.1. AMBIENTE DE DESENVOLVIMENTO 8
2.2. MODELOS DE PROGRAMAÇÃO 8
2.3. CRIANDO UM NOVO PROJETO 9
3. CONVERSOR ANALÓGICO DIGITAL (ADC) 11
3.1. VISÃO GERAL 11
3.2. REFERENCE VOLTAGE GENERATOR 12
3.3. INPUT CIRCUIT 12
3.4. SOC0-SOC15 CONFIGURATIONS 13
3.5. ADC INTERRUPT LOGIC 14
3.6. PRIORIDADE DAS CONVERSÕES 15
3.7. DEMAIS FUNÇÕES 17
3.8. EXEMPLOS DE CÓDIGOS UTILIZANDO O ADC 18
4. MÓDULO EPWM 29
4.1. VISÃO GERAL 29
4.2. TIME-BASE (TB) 32
4.3. COUNTER-COMPARE (CC) 34
4.4. ACTION-QUALIFIER (AQ) 36
4.5. DEAD-BAND (DB) 38
4.6. PWM-CHOPPER (PC) 39
4.7. TRIP-ZONE (TZ) 39
4.8. EVENT-TRIGGER (ET) 40
4.9. DIGITAL COMPARE (DC) 42
4.10. EXEMPLO DE CÓDIGO UTILIZANDO O EPWM 42
5 CONCLUSÃO 49
REFERÊNCIAS BIBLIOGRÁFICAS 50
ANEXO 1 51
ANEXO 2 55
ANEXO 3 60

1. INTRODUÇÃO

Microcontroladores de propósito geral e *DSPs* apresentam várias características em comum. De fato, em relação à sua arquitetura, são extremamente semelhantes. As diferenças aparecem quando lidamos com suas capacidades de processamento de sinais. Os microcontroladores são circuitos digitais desenvolvidos para trabalharem tanto com manipulação de dados quanto para cálculos matemáticos. Por serem dispositivos genéricos, deve-se haver um compromisso entre essas duas aplicações, de forma que a estrutura projetada para atender a estas funcionalidades não pode ser otimizada para ambas. Isto dificulta sua aplicação em sistemas em tempo real, onde a velocidade de cálculo é um parâmetro fundamental.

Os DSPs, por outro lado, são dispositivos concebidos com foco no processamento. Seus circuitos são projetados de forma a otimizarem o tempo de cálculo nas operações matemáticas, bem como realizá-las em um tempo previsível e contínuo, o que permite o processamento de grandes quantidades de dados. Em geral, as operações mais recorrentes como adição e multiplicação são realizadas antecipadamente e armazenados em acumuladores, de forma que um número maior de cálculos é feito em um único ciclo de processamento.

O kit de desenvolvimento *Piccolo Launchpad* é baseado no microcontrolador *TMS320F28027*, produzido pela *Texas Instruments*. Ele pertence à família *Piccolo* da linha *C2000*, linha de microcontroladores para controle em tempo real. Ele não se trata, portanto, de um DSP, mas de um microcontrolador de baixo custo. Entretanto, ele nos oferece uma série de recursos que nos permitem realizar diversas aplicações em tempo real.

Neste trabalho, desejamos fornecer um ponto de partida para o desenvolvimento de aplicações no Piccolo Launchpad. Ele será estruturado como um tutorial. Os primeiros tópicos apresentam as características do F28027, o ambiente de desenvolvimento *Code Composer Studio*, os modelos de programação e o passo-a-passo para a criação de novos projetos. Em seguida, abordaremos os periféricos *ADC (Analog Digital Converter)* e *ePWM (Enhanced Pulse Width*

Modulator), explicando seus princípios de funcionamento e dando exemplos de códigos comentados, de forma a facilitar o entendimento por parte do leitor.

2. CARACTERÍSTICAS DO MICROCONTROLADOR

O microcontrolador TMS320F28027 apresenta as seguintes especificações:

- Arquitetura de 32 bits;
- Clock de 60MHz;
- ADC de 12 bits (16 canais multiplexados e 2 circuitos Sample and Hold);
- 8 módulos PWM com timers de 16 bits independentes;
- Módulo PWM de alta resolução;
- 3 CPU Timers;
- 22 pinos de GPIO (General Purpose Input/Output);
- Comunicação I2C, SPI, UART;
- 64KB de memória flash;

O kit reúne todo o hardware e software necessários para realizar aplicações baseadas no F28027 de maneira simples. A ferramenta *JTAG* realiza a interface entre o microcontrolador e um computador pessoal para uma fácil programação, debug e testes. Além disso, ele conta com uma interface de comunicação serial *UART*, que se comunica através de uma porta USB.

A placa apresenta quatro barramentos de dez pinos que se conectam aos pinos do microcontrolador, quatro LEDs (acesos em nível lógico baixo) e duas chaves push-bottom. Isto facilita a interface entre o microcontrolador e o desenvolvedor na realização de testes. Utilizaremos estes recursos quando apresentarmos os exemplos.

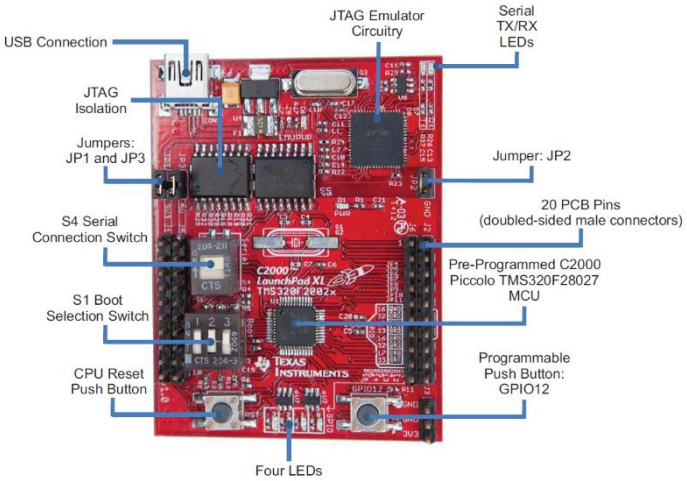


Figura 0.1: Launchpad C200.

2.1. AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento integrado (IDE) recomendado pela Texas Instruments é o Code Composer Studio. Neste trabalho, utilizamos a versão 6.1.3, disponibilizada gratuitamente no site da Texas Instruments [1] – é necessário, entretanto, criar uma conta no site. Este IDE é foi desenvolvido pela própria TI e oferece diversas ferramentas para desenvolvimento e emulação em microcontroladores e DSPs.

Juntamente ao Code Composer, utilizamos o software *ControlSuite*, que contém diversos drivers, manuais, bibliotecas e exemplos da maior parte dos dispositivos fabricados pela Texas, incluindo documentação específica do Piccolo Launchpad. Ele é extremamente útil, principalmente aos iniciantes.

É necessário que o leitor instale os dois softwares, cujos links são dados em [1] e [2].

2.2. MODELOS DE PROGRAMAÇÃO

A Texas Instruments disponibiliza no Code Composer os drivers necessários ao desenvolvimento no TMS320F28027. Ele oferece dois modelos de programação: *Direct Register Acces Model* (DRA) e *Software Driver Model* (SD). Os dois modelos podem ser utilizados independentemente ou combinados.

O modelo DRA trabalha em uma camada inferior ao SD, sendo necessário, portanto, um conhecimento acerca da arquitetura do dispositivo a nível de registradores. A linguagem de programação utilizada é também a linguagem C. A vantagem desse modelo é a eficiência e o tamanho do código. Em aplicações que requerem um código mais otimizado, este modelo é o recomendável.

O modelo SD utiliza estruturas (handles) e tipos enumerados que abstraem o acesso aos registradores, tornando o desenvolvimento mais simples e códigos mais legíveis. É recomendado para aplicações que não necessitam de código e tempo de execução ótimos. Podemos ainda utilizar ambos os modelos, caso a eficiência de código não seja um problema mas o tempo de execução seja um parâmetro crítico.

Ao longo do texto apresentaremos os dois modelos, mas o foco será dado no modelo Software Driver. O leitor que deseja saber mais sobre o modelo Direct Register Acess poderá consultar a documentação de cada periférico, bem como o documento [3].

2.3. CRIANDO UM NOVO PROJETO

O processo de criação de um projeto, cujo código será gravado no launchpad, requer algumas configurações não triviais. Para simplificar essa tarefa, iremos importar o exemplo do sensor de temperatura, o mesmo que vem pré-programado no launchpad e disponível no ControlSuite.

Os passos para a importação do exemplo são dados abaixo.

1) Executar o software Code Composer Studio 6.1.3. Caso seja a primeira vez, o programa solicita a escolha de um diretório para a criação de um workspace;

2) Na página inicial *Getting Started*, clique em *Browser Examples*. A aba *TI Resource Explorer* se abrirá;

3) No canto esquerdo, clique em *controlSUITE -> English -> Development Tools -> C2000 LaunchPad -> LAUNCHXL-F28027 -> C2000 LaunchPad Demo*;

4) Siga os passos informados no *TI Resource Explorer* (veja figura abaixo). O último passo não é necessário, pois se trata da gravação do exemplo na placa;

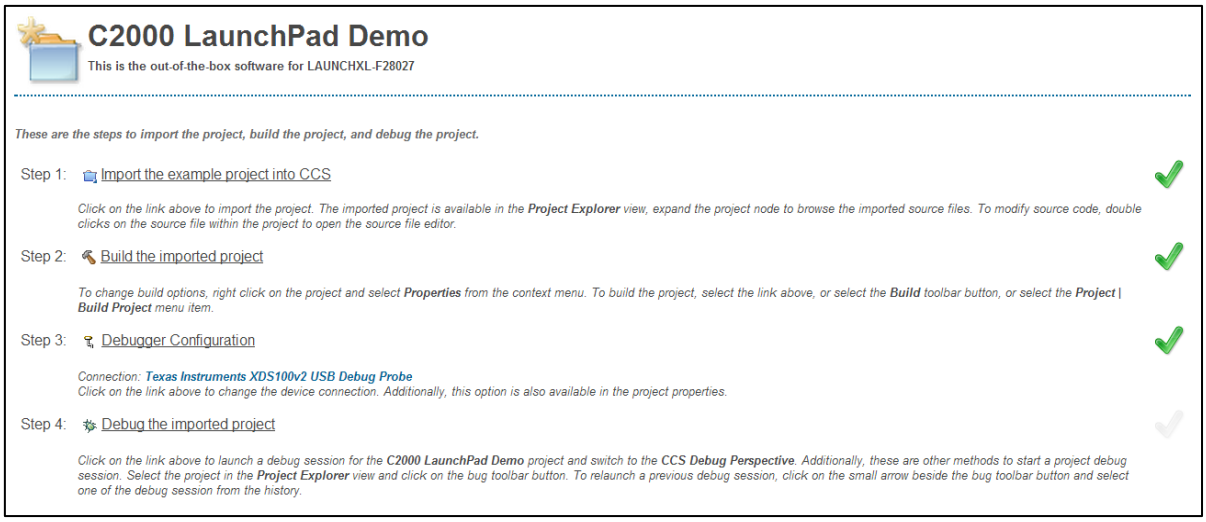


Figura 0.2: TI Ressource Explorer.

5) Na aba Project Explorer, selecione o projeto que acaba de ser importado e dê duplo clique no arquivo Example_F2802xLaunchPadDemo.c. Este é o arquivo que iremos inserir nosso código exemplo.

Após inserir o código neste arquivo, salve o arquivo, clique em *Build* para construir o projeto e *Debug* para executá-lo no Launchpad.

3. CONVERSOR ANALÓGICO DIGITAL (ADC)

3.1. VISÃO GERAL

O TMS320F28027 possui um único conversor analógico/digital de 12 bits e dois circuitos *Sample and Hold* (S&H), que permitem a aquisição simultânea ou sucessiva de dois canais. Ele oferece um total de 16 canais de entrada multiplexados, sendo divididos nos blocos A e B. Cada bloco é conectado a um circuito S&H. Portanto, caso deseja-se realizar aquisições simultâneas, deve-se utilizar um canal de cada bloco.

Neste ADC, uma conversão pode ser iniciada por diferentes fontes (triggers): disparo por software (S/W), módulos ePWM, entradas GPIO, CPU timers ou interrupções do próprio ADC. Ao fim das conversões, ele possui 9 diferentes bits de interrupções que podem ser geradas na CPU.

A figura abaixo nos mostra o diagrama em blocos do conversor A/D, que será descrito em detalhes.

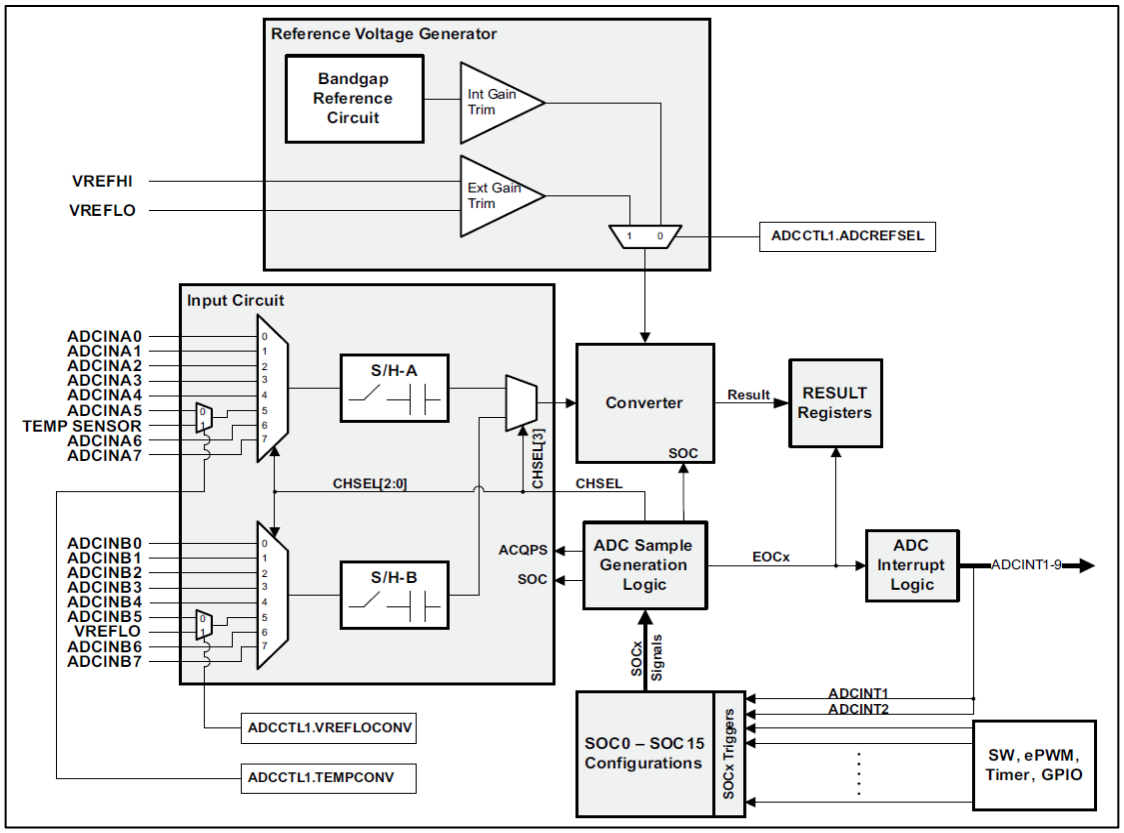


Figura 3.1: Diagrama em blocos do ADC.

OBS: O launchpad conecta apenas 13 dos 16 canais, não sendo possível utilizar os canais ADCINA5, ADCINB0 e ADCINB5. Porém, podemos utilizar o sensor de temperatura que é conectado internamente e multiplexado com o canal ADCINA5.

3.2. REFERENCE VOLTAGE GENERATOR

O bloco Reference Voltage Generator é o responsável pelas tensões de referência do conversor. A referência da medição pode escolhida internamente, sendo a faixa fixa de 0V a 3,3V ou externamente. Escolhendo-se a referência externa, inserimos uma tensão de referência superior (VREFHI), inferior (VREFLO) ou ambas, dentro da faixa de 0V a 3,3V. Contudo, o launchpad mantém fixa em GND a entrada VREFLO, sendo possível inserir apenas o limite superior. A resolução é determinada dividindo-se os limites pelos 12 bits do conversor (ou seja, 4096 possíveis valores). Para valores de tensão medidos inferiores à 0V, o ADC armazena o valor 0 e para valores superiores a 3,3V, armazena o valor 4095.

O registrador ADCCTL1.ADCREFSEL define se a referência é interna (0) ou externa (1). Escolhida a referência externa, utilizamos o pino 6 do jumper 5 do kit para inserir o valor de tensão VREFHI. Esta entrada é compartilhada com a primeira entrada do bloco A do conversor (ADCINA0), portanto, perdemos um canal ao se utilizar a referência externa.

3.3. INPUT CIRCUIT

O circuito de entrada é formado pelos canais de entrada, dividos nos blocos A e B, cada um conectado a um S&H. As entradas são multiplexadas, bem como os circuitos S&H. Após a conversão, o valor convertido é armazenado em um registrador ADCRESULT. Ao todo, temos 16 registradores ADCRESULT, um para cada SOC. Se, por exemplo, o SOC6 é configurado com determinado canal, trigger e SW, o resultado da conversão é armazenado em ADCRESULT6.

Notem que temos ainda um sensor de temperatura do chip que compartilha o mesmo pino do canal ADCINA5. Podemos utilizá-lo, caso seja necessário, para o controle térmico do chip.

3.4. SOC0-SOC15 CONFIGURATIONS

Este bloco trata de um importante aspecto do conversor A/D. O ADC é dito do tipo *SOC based,* pois é baseado em configurações de blocos denominados SOCs (star-of-convertions). Ao todo, temos 16 SOCs, que são utilizados independentemente para associar uma entrada simples, um trigger e uma janela de amostragem.

Essas três informações podem ser associadas de forma que qualquer dos 16 canais pode ser utilizado com qualquer dos triggers disponíveis e qualquer das janelas de amostragem. Podemos por exemplo associar múltiplos SOCs ao mesmo canal, por exemplo o ADCINA2, com diferentes triggers e janelas de amostragem, ou selecionar o mesmo trigger para disparar todos os canais (prioridades, neste caso, serão tratadas posteriormente) com a mesma ou diferentes janelas de amostragem. Esta característica nos permite uma grande flexibilidade para o uso do conversor, permitindo uma série de diferentes conversões em tempos bem definidos.

A figura abaixo apresenta um diagrama em blocos detalhado do princípio de operação SOC. A configuração é realizada através de registradores de controle que são descritos em [4].

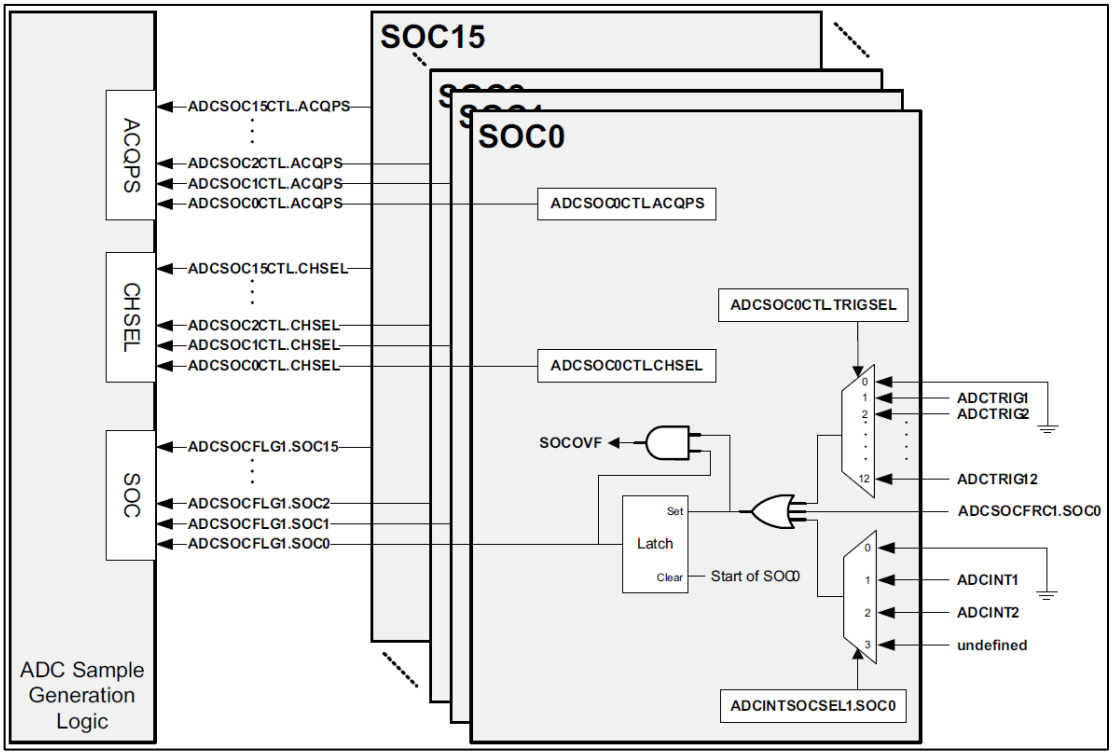


Figura 3.2: Diagrama do bloco SOC.

Janelas de Amostragem

As entradas do conversor A/D podem ser modeladas por circuitos RC. Alguns circuitos externos que são conectados às entradas do ADC necessitam de um tempo maior para realizar a transferência de carga para os seus elementos armazenadores. Por esta razão, o ADC do F28027 nos oferece a possibilidade de ajustar a janela de amostragem (ACQPS) para cada SOC individualmente.

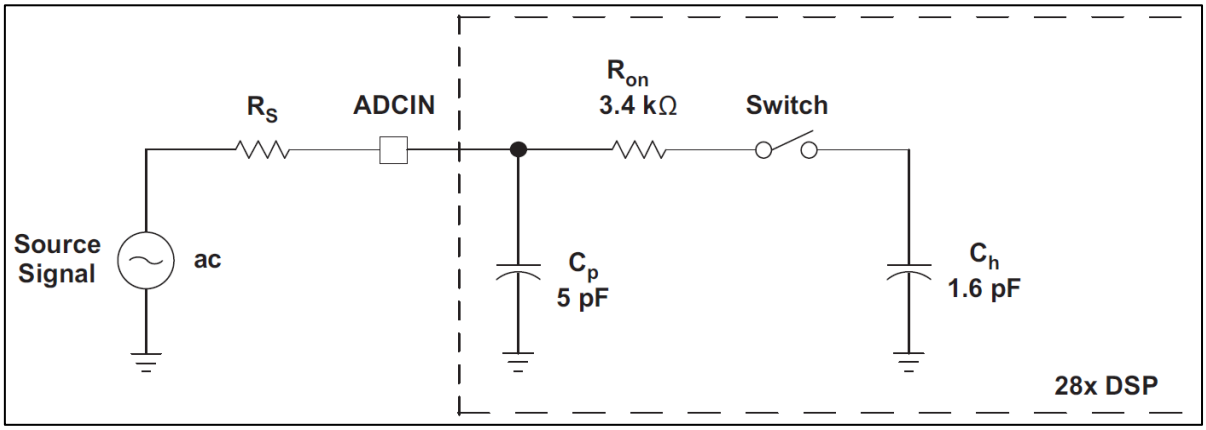


Figura 3.3: Circuito equivalente de entrada do ADC.

O valor da janela de amostragem pode ser escolhida entre 7 e 26 ciclos de clock (no nosso caso, sendo o clock de 60MHz, entre 116,67ns e 433,67ns). É importante mencionar que no nosso dispositivo, ajustamos o valor campo ACQPS do registrador de controle com uma unidade a menos que a janela de amostragem desejada. Se queremos por exemplo 7 ciclos de clock, fazemos ACQPS = 6.

3.5. ADC INTERRUPT LOGIC

Para cada SOC, temos associado um EOC (end-of-convertion). Ao todo, os 16 EOCs são configurados individualmente para gerarem pulsos no início ou no final de cada conversão. O ADC possui 9 bits interrupções disponíveis na PIE (Peripheral Interrupt Expansion) e podem ser selecionadas para receberem os pulsos dos EOCs.

Segue abaixo uma figura que apresenta a estrutura do bloco de interrupções.

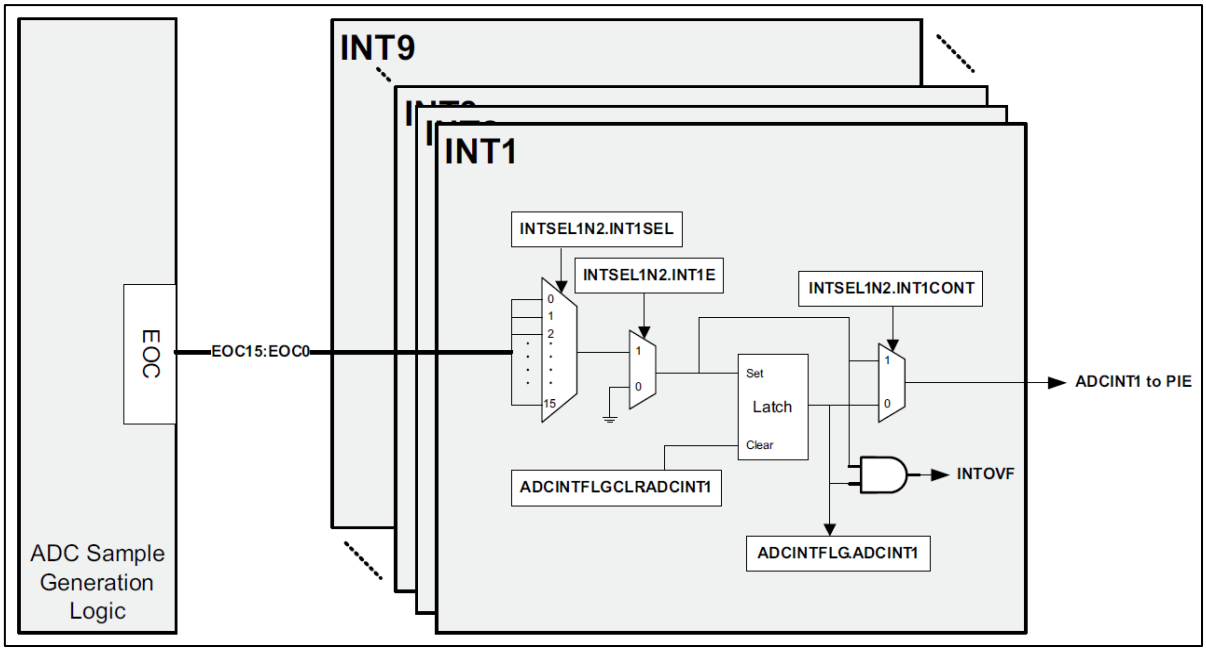


Figura 3.4: Diagrama do bloco de interrupções.

3.6. PRIORIDADE DAS CONVERSÕES

É importante determinar a ordem em que as conversões são realizadas para o caso em que o mesmo trigger seja associado a múltiplas entradas do ADC, ou se diferentes fontes de trigger disparam conversões no mesmo instante.O critério padrão de escolha é determinado pelo método Round Robin.

Supomos que os 16 SOCs do ADC são dispostos ordenadamente na forma de círculo. Temos um ponteiro (RRPOINTER) que aponta para SOC0. No caso de disparos simultâneos, a precedência é determinada pela ordem da roda. Supondo que o último canal disparado seja SOC8, então o ponteiro será apontado para ele após a conversão. Um novo disparo simultâneo, portanto, seguirá a ordem da roda à partir de SOC8. As figuras abaixo retiradas de [4] nos fornecem mais exemplos que ilustram o método.

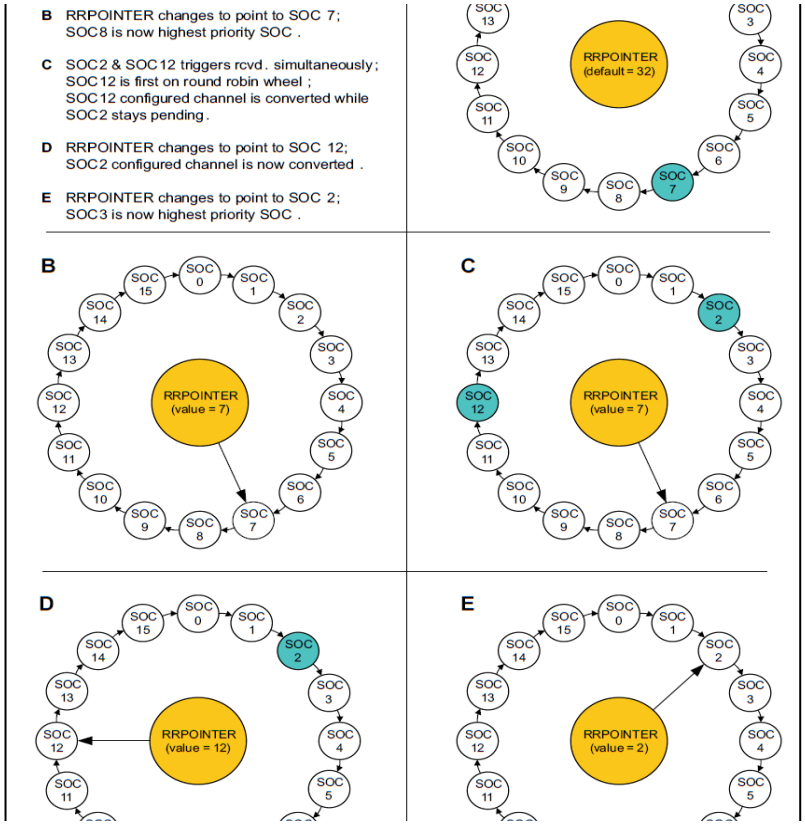


Figura 3.5: Método Round Robin; Exemplo 1.

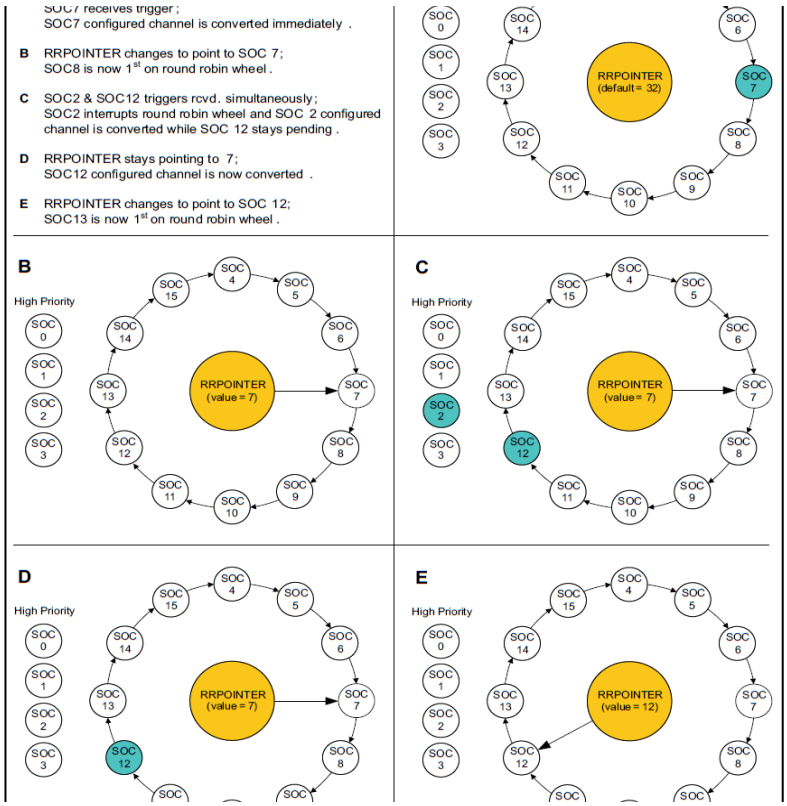


Figura 3.6: Método Round Robin; Exemplo 2.

3.7. DEMAIS FUNÇÕES

Muitos outros recursos como aquisição simultânea, conversão ONESHOT ou calibração são explicados detalhadamente em [4], mas não serão mencionados neste trabalho. Os recursos já descritos são essenciais para o uso do conversor e , após tratados todos estes tópicos, estamos aptos a entender os atributos e métodos dos handles que utilizaremos na programação SDM do ADC. O leitor que deseja programar com acesso direto aos registradores (DRA) deve consultar o documento [4] que contém todas as informações sobre os registradores do ADC e do Comparador Analógico do F28027.

3.8. EXEMPLOS DE CÓDIGOS UTILIZANDO O ADC

O documento [3] nos informa todos os atributos e métodos dos periféricos do microcontrolador TMS320F28027. No Code Composer, pressionando o botão CTRL e clicando em alguma função, seu arquivo de cabeçalho (.h) é aberto. Geralmente os arquivos de cabeçalho apresentam uma breve descrição da função, bem como seus parâmetros. Os tipos enumerados de cada periféricos também se encontram nestes arquivos e são imprescindíveis para a programação SD.

EXEMPLO 1 - CONVERSÃO DISPARADA POR PWM

Neste exemplo, utilizamos a EPWM1 como trigger dos canais ADCINA2, conectado ao pino 8 do J1, e ADCINA4, conectado ao pino 6 do mesmo jumper. As conversões são iniciadas com período constante e os resultados das conversões são comparados. Se a tensão no pino 8 for superior à do pino 6**,** o LED 0 apaga e o LED 1 acende. Do contrário, o LED 0 acende e o LED 1 apaga.

Iniciamos o código com a inclusão da biblioteca do dispositivo. Todos os programas devem ser iniciados por este comando.

```
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo
```

Em seguida, incluímos as bibliotecas de cada periférico que será utilizado. As comentadas com um asterisco sempre – ou quase sempre – estarão presentes, pois fazem parte da configuração inicial do dispositivo.

```
#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock*
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH*
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE *
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL*
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog*
```

Em seguida, declaramos a rotina de interrupção que o ADC executará, bem como as variáveis globais que utilizaremos.

```
__interrupt void adc_isr(void); //Protótipo da rotina de interrupção

// Variáveis globais utilizadas nesse exemplo:
uint16_t LoopCount;
uint16_t Voltage6; //Tensão no pino 6
uint16_t Voltage8; //Tensão no pino 8
```

Declaramos também os *handles*, que são as estruturas disponíveis afim de abstrair a configuração e utilização dos periféricos, uma das vantagens mais importantes do modelo SD. Os handles globais são declarados no escopo, e os locais dentro da função principal.

```
// Declara os handles globais:
ADC_Handle myAdc;
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm;
```

Declaramos e implementamos a função que configura o GPIO. Lembrando que em C podemos codificar a função ao final do programa, mas sua declaração deve constar no topo.

```
void Gpio_setup() // Configura as saídas conectadas aos LEDs no GPIO
{

    // Habilita resistores de pull-up:
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Enable);

    // Seta os pinos no modo “propósito geral”:
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_GeneralPurpose);

    // Define os pinos como saídas
    GPIO_setDirection(myGpio, GPIO_Number_0, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_1, GPIO_Direction_Output);

    // Seta as saídas em nível lógico baixo, o que apaga os LEDs:
    GPIO_setHigh(myGpio, GPIO_Number_0);
    GPIO_setHigh(myGpio, GPIO_Number_1);
}
```

Iniciamos a função principal *main.c* declarando os *handles* que são utilizados localmente e, em seguida, iniciando cada um deles. Em geral, os comandos de inicialização mostrados abaixo são sempre os mesmos.

```
void main(void)
{
    // Declara os handles locais:
    CPU_Handle myCpu;
    PLL_Handle myPll;
    WDOG_Handle myWDog;

    // Inicializa todos os handles da aplicação:
    myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));
    myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
    myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
    myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
    myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
    myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
    myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
    myPwm = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
    myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));
```

A sequência abaixo se trata da configuração inicial do dispositivo que, para nossas aplicações, serão sempre as mesmas. Não entraremos em detalhes sobre cada linha, mas fica evidente que eles se tratam das configurações relacionadas ao oscilador do dispositivo, PLL, Watchdog e outros periféricos.

```
    // Executa a inicialização básica do sistema:
    WDOG_disable(myWDog);
    CLK_enableAdcClock(myClk); // Habilita o clock do ADC
    (*Device_cal)();

    //Seleciona o oscilador interno 1 para ser o clock:
    CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

    // Configura a PLL para x12 / 2 que resulta em  60Mhz = 10Mhz * 12 / 2:
    PLL_setup(myPll, PLL_Multiplier_12, PLL_DivideSelect_ClkIn_by_2);

    // Desabilita a PIE e todas as interrupções, e limpa os flags:
    PIE_disable(myPie);
    PIE_disableAllInts(myPie);
    CPU_disableGlobalInts(myCpu);
    CPU_clearIntFlags(myCpu);

    // [Usado no modo de configuração "FLASH" do CC] Copia funções da RAM para a RAM:
    #ifdef _FLASH
        memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
    #endif

    // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
    PIE_setDebugIntVectorTable(myPie); // *Somente no modo de configuração DEBUG
    PIE_enable(myPie);
```

O próximo comando é importante pois seleciona uma entre as 9 interrupções disponíveis no ADC. Os grupos e subgrupos selecionados são os referentes à interrupção *ADCINT1* da PIE [5].

```
// Configura a rotina de interrupção do ADCINT1 na PIE:
PIE_registerPieIntHandler(myPie, PIE_GroupNumber_10, PIE_SubGroupNumber_1,
(intVec_t)&adc_isr);
```

Em seguida, iniciamos o ADC. Estas configurações dizem respeito à funcionalidades que não foram abordadas aqui, portanto as utilizaremos sempre desta forma. Para mais detalhes, ver documento [4].

```
// Inicialização básica do ADC:
ADC_enableBandGap(myAdc); // Habilita o circuito da tensão de referência
ADC_enableRefBuffers(myAdc); // Habilita o buffer do circuito de referência
ADC_powerUp(myAdc); // Energiza os circuitos do ADC (ver tópico Power Up Sequence em [4])
ADC_enable(myAdc); // Habilita o ADC (ver tópico Power Up Sequence em [4])
ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int); // Seleciona a referência interna
```

Ao grupo 10 e subgrupo 1 da tabela de interrupções (PIE), está associada a interrupção ADCINT1. As próximas funções habilitam as interrupções no PIE e na CPU.

```
PIE_enableAdcInt(myPie, ADC_IntNumber_1); // Habilita ADCINT1 na PIE

CPU_enableInt(myCpu, CPU_IntNumber_10); // Habilita CPU Interrupt 10 (grupo 10 da PIE -
PIE_GroupNumber_X -> CPU_IntNumber_X)

CPU_enableGlobalInts(myCpu); // Habilita a interrupção global INTM (ver [5])

CPU_enableDebugInt(myCpu); // Habilita a interrupção global do modo de emulação em tempo
real DBGM (ver [5])
```

O próximo passo é configurar os canais do ADC, bem como os SOCs. Um detalhe importante a ser mencionado é o *workaround* provido em [6] que diz respeito à primeira amostra. Devido a um problema no dispositivo, a primeira de uma série de conversões simultâneas ou sequenciais disparadas por determinado trigger não é realizada corretamente. Para isso, recomenda-se configurar um SOC qualquer a ser colocado no início das conversões. Em seguida, ignoramos o resultado desta primeira conversão e utilizamos as demais. Neste exemplo, configuramos o SOC0, cujo resultado será descartado.

// Configuração do ADC
//OBS: O canal ADCINA4 será associado a dois SOCs para ser duplamente amostrado, como workaround provido no documento rev0 silicon errata

ADC_setIntPulseGenMode(myAdc, ADC_IntPulseGenMode_Prior); *//Configura o pulso da interrupção ADCINT1 para ocorrer no fim da conversão*

ADC_enableInt(myAdc, ADC_IntNumber_1); *//Habilita a interrupção ADCINT1*

ADC_setIntMode(myAdc, ADC_IntNumber_1, ADC_IntMode_ClearFlag); *//Seleciona ADCINT1 para o modo Clear Flag (uma nova interrupção não pode ser gerada enquanto o flag não for resetado)*

ADC_setIntSrc(myAdc, ADC_IntNumber_1, ADC_IntSrc_EOC2); *//Configura EOC2 (associado a SOC2) para disparar ADCINT1*

ADC_setSocChanNumber (myAdc, ADC_SocNumber_0, ADC_SocChanNumber_A4);
// associa SOC0 no canal ADCINA4

ADC_setSocChanNumber (myAdc, ADC_SocNumber_1, ADC_SocChanNumber_A4);
// associa SOC1 no canal ADCINA4

ADC_setSocChanNumber (myAdc, ADC_SocNumber_2, ADC_SocChanNumber_A2);
// associa SOC2 no canal ADCINA2

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_0, ADC_SocTrigSrc_EPWM1_ADCSOCA);
//seleciona EPWM1A para disparar SOC0 (devido ao round-robin, SOC0 é amostrado antes de SOC1

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_1, ADC_SocTrigSrc_EPWM1_ADCSOCA);
// seleciona EPWM1A para disparar SOC1 (devido ao round-robin, SOC0 é amostrado antes de SOC1

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_2, ADC_SocTrigSrc_EPWM1_ADCSOCA);
// seleciona EPWM1A para disparar SOC2 (devido ao round-robin, SOC1 é amostrado antes de SOC2

ADC_setSocSampleWindow(myAdc, ADC_SocNumber_0,
ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC0 para 7 ciclos de clock, (6 ACQPS + 1)*

ADC_setSocSampleWindow(myAdc, ADC_SocNumber_1,
ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC1 para 7 ciclos de clock, (6 ACQPS + 1)*

ADC_setSocSampleWindow(myAdc, ADC_SocNumber_2,
ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC2 para 7 ciclos de clock, (6 ACQPS + 1)*

Devemos agora configurar o EPWM, uma vez que ele será o trigger dos canais do ADC. Este trecho do código ficará mais claro quando abordamos o módulo PWM.

// Habilita o clock do módulo PWM1:
CLK_enablePwmClock(myClk, PWM_Number_1);

```
    // Configura  ePWM1:
    PWM_enableSocAPulse(myPwm); // Habilita a geração de pulsos para SOCs (grupo A)

    PWM_setSocAPulseSrc(myPwm,  PWM_SocPulseSrc_CounterEqualCmpAIncr);  //  Seleciona
    a condição para geração do pulso. Nesse caso, o disparo ocorrerá quando o valor do contador da
    PWM for igual ao valor do comparador A na subida

    PWM_setSocAPeriod(myPwm, PWM_SocPeriod_FirstEvent); // Define que o pulso será gerado
    no primeiro evento

    PWM_setCmpA(myPwm, 0x0080); // Seta o comparador com o valor 0x0080

    PWM_setPeriod(myPwm, 0xFFFF); // Seta o período com o valor 0xFFFF

    PWM_setCounterMode(myPwm,  PWM_CounterMode_Up);  //  Define  o  modo  de  contagem
    progressivo

    //Habilita o clock do time-base
    CLK_enableTbClockSync(myClk);
```

O restante da função *main.c* segue abaixo.

```
    Gpio_setup(); // Chama a função de configuração da GPIO

    LoopCount = 0;

     // Aguarda a interrupção do ADC
    for(;;)
    {
        LoopCount++;
    }
}
```

Por fim, devemos implementar a rotina de interrupção do ADC, que será responsável por recuperar os valores das conversões, executar as ações desejadas e resetar os flags de interrupção.

```
__interrupt void adc_isr(void) // Rotina de interrupção
{
    //Devemos descartar o resultado da primeira conversão (SOC0) devido ao workaround de [rev0
    silicon errata]:

    Voltage6 = ADC_readResult(myAdc, ADC_ResultNumber_1); // Recupera o valor da conversão
    do canal ADCINA4, associado à SOC1

    Voltage8 = ADC_readResult(myAdc, ADC_ResultNumber_2); // Recupera o valor da conversão
    do canal ADCINA2, associado à SOC2

    if(Voltage6 < Voltage8) // Compara os valores das tensões
    {
        // Acende o LED1 se a tensão no pino 8 é superior à do pino 6
        GPIO_setHigh(myGpio, GPIO_Number_0);
        GPIO_setLow(myGpio, GPIO_Number_1);
```

```
    }
    else
    {
        // Acende o LED0 caso contrário
        GPIO_setLow(myGpio, GPIO_Number_0);
        GPIO_setHigh(myGpio, GPIO_Number_1);
    }

    // Limpa o flag de ADCINT1 para o próximo evento
    ADC_clearIntFlag(myAdc, ADC_IntNumber_1);

    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_10);
}
```

No Anexo 1 encontra-se o código condensado, de modo que o leitor possa copiá-lo no Code Composer Studio e executá-lo.

EXEMPLO 2 - CONVERSÃO DISPARADA POR TIMER

O código a ser apresentado realiza a conversão de dois sinais analógicos conectados aos mesmos canais do exemplo anterior (ADCINA2, conectado ao pino 8 do J1, e ADCINA4, conectado ao pino 6 do mesmo jumper). O trigger que dispara as conversões é o Cpu-Timer 0.

Os LEDs 0 e 1 são acessos nas mesmas condições do exemplo 1. Acrescentamos os LEDs 2 e 3 para indicar o momento do disparo. Utilizamos, portanto, a função de interrupção do Cpu-timer 1 para acender e apagar alternadamente estes dois LEDs. O mesmo evento que chama a rotina de interrupção do timer é o que dispara as conversões do ADC. A ADC é programado para gerar sua interrupção ao fim da conversão de SOC2, associado à entrada ADCINA4.

Não detalharemos cada etapa do código como foi feito no Exemplo 1, pois a maioria das configurações são exatamente as mesmas.

Definimos as bibliotecas.

```
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo

#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
```

```
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog
#include "f2802x_common/include/timer.h" // inclui biblioteca do cpu-timer
```

Incluímos uma definição para o período, que deve ser selecionado pelo usuário. O valor é o tempo em microsegundos. Em seguida, declaramos as rotinas de interrupção, variáveis globais e handles.

```
#define period 1000000 // Período em microsegundos

// Protótipo das rotinas de interrupção
__interrupt void adc_isr(void); //Protótipo da rotina de interrupção do ADC
__interrupt void cpu_timer0_isr(void); //Protótipo da rotina de interrupção do cpu-timer0

// Variáveis globais utilizadas nesse exemplo:
uint16_t Voltage6; //Tensão no pino 6
uint16_t Voltage8; //Tensão no pino 8

// Declara os handles globais:
ADC_Handle myAdc;
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm;
```

Definimos e implementamos a rotina da configuração das saídas GPIO.

```
void Gpio_setup() // Configura as saídas conectadas aos LEDs no GPIO
{
    // Habilita resistores de pull-up:
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_2, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_3, GPIO_PullUp_Enable);

    // Seta os pinos no modo “propósito geral”:
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_2, GPIO_2_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_3, GPIO_3_Mode_GeneralPurpose);

    // Define os pinos como saídas
    GPIO_setDirection(myGpio, GPIO_Number_0, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_1, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_2, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_3, GPIO_Direction_Output);

    // Define o nível lógico inicial dos LEDs:
    GPIO_setHigh(myGpio, GPIO_Number_0);
    GPIO_setHigh(myGpio, GPIO_Number_1);
    GPIO_setHigh(myGpio, GPIO_Number_2);
    GPIO_setLow(myGpio, GPIO_Number_3);
}
```

Começamos a função principal com a mesma configuração do exemplo anterior. Incluímos o registro da função de interrupção do timer na PIE.

```
void main(void)
{
    // Declara os handles locais:
    CPU_Handle myCpu;
    PLL_Handle myPll;
    WDOG_Handle myWDog;
    TIMER_Handle myTimer0;

    // Inicializa todos os handles da aplicação:
    myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));
    myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
    myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
    myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
    myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
    myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
    myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
    myPwm = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
    myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));
    myTimer0 = TIMER_init((void *)TIMER0_BASE_ADDR, sizeof(TIMER_Obj));

    // Executa a inicialização básica do sistema:
    WDOG_disable(myWDog);
    CLK_enableAdcClock(myClk); // Habilita o clock do ADC
     (*Device_cal)();

    //Seleciona o oscilador interno 1 para ser o clock:
    CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

    // Configura a PLL para x12 / 2 que resulta em  60Mhz = 10Mhz * 12 / 2:
    PLL_setup(myPll, PLL_Multiplier_12, PLL_DivideSelect_ClkIn_by_2);

    // Desabilita a PIE e todas as interrupções, e limpa os flags:
    PIE_disable(myPie);
    PIE_disableAllInts(myPie);
    CPU_disableGlobalInts(myCpu);
    CPU_clearIntFlags(myCpu);

    // [Usado no modo de configuração "FLASH" do CC] Copia funções da RAM para a RAM:
    #ifdef _FLASH
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
    #endif

    // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
    PIE_setDebugIntVectorTable(myPie);
    PIE_enable(myPie);

    // Configura as rotinas de interrupção na PIE:
    PIE_registerPieIntHandler(myPie,        PIE_GroupNumber_10,        PIE_SubGroupNumber_1,
    (intVec_t)&adc_isr);
    PIE_registerPieIntHandler(myPie,        PIE_GroupNumber_1,        PIE_SubGroupNumber_7,
    (intVec_t)&cpu_timer0_isr);

    // Inicialização básica do ADC:
    ADC_enableBandGap(myAdc); // Habilita o circuito da tensão de referência
```

ADC_enableRefBuffers(myAdc); *// Habilita o buffer do circuito de referência*
ADC_powerUp(myAdc); *// Energiza os circuitos do ADC (ver tópico Power Up Sequence em [4])*
ADC_enable(myAdc); *// Habilita o ADC (ver tópico Power Up Sequence em [4])*
ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int); *// Seleciona a referência interna*

Habilitamos as interrupções, inclusive do Cpu-timer 0.

PIE_enableAdcInt(myPie, ADC_IntNumber_1); *// Habilita ADCINT1 na PIE*

PIE_enableTimer0Int(myPie); *// Habilita timer0 na PIE*

CPU_enableInt(myCpu, CPU_IntNumber_10); *// Habilita CPU Interrupt 10 (grupo 10 da PIE - PIE_GroupNumber_X -> CPU_IntNumber_X; ADC)*

CPU_enableInt(myCpu, CPU_IntNumber_1); *// Habilita CPU Interrupt 1 (timer)*

CPU_enableGlobalInts(myCpu); *// Habilita a interrupção global INTM (ver [5])*

CPU_enableDebugInt(myCpu); *// Habilita a interrupção global do modo de emulação em tempo real DBGM (ver [5])*

Configuramos o ADC da mesma maneira. A única diferença é o trigger dos SOCs, que são associados ao Cpu-timer 0.

// Configuração do ADC
//OBS: O canal ADCINA4 será associado a dois SOCs para ser duplamente amostrado, como workaround provido no documento rev0 silicon errata

ADC_setIntPulseGenMode(myAdc, ADC_IntPulseGenMode_Prior); *//Configura o pulso da interrupção ADCINT1 para ocorrer no fim da conversão*
ADC_enableInt(myAdc, ADC_IntNumber_1); *//Habilita a interrupção ADCINT1*
ADC_setIntMode(myAdc, ADC_IntNumber_1, ADC_IntMode_ClearFlag); *//Seleciona ADCINT1 para o modo Clear Flag (uma nova interrupção não pode ser gerada enquanto o flag não for resetado)*
ADC_setIntSrc(myAdc, ADC_IntNumber_1, ADC_IntSrc_EOC2); *//Configura EOC2 (associado a SOC2) para disparar ADCINT1*

ADC_setSocChanNumber (myAdc, ADC_SocNumber_0, ADC_SocChanNumber_A4); *// associa SOC0 no canal ADCINA4*
ADC_setSocChanNumber (myAdc, ADC_SocNumber_1, ADC_SocChanNumber_A4); *// associa SOC1 no canal ADCINA4*
ADC_setSocChanNumber (myAdc, ADC_SocNumber_2, ADC_SocChanNumber_A2); *// associa SOC2 no canal ADCINA2*

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_0, ADC_SocTrigSrc_CpuTimer_0); *//seleciona cpu_timer0 para disparar SOC0 (devido ao round-robin, SOC0 é amostrado antes de SOC1)*
ADC_setSocTrigSrc(myAdc, ADC_SocNumber_1, ADC_SocTrigSrc_CpuTimer_0); *// seleciona cpu_timer0 para disparar SOC1 (devido ao round-robin, SOC0 é amostrado antes de SOC1)*
ADC_setSocTrigSrc(myAdc, ADC_SocNumber_2, ADC_SocTrigSrc_CpuTimer_0); *// seleciona cpu_timer0 para disparar SOC2 (devido ao round-robin, SOC1 é amostrado antes de SOC2)*

```
    ADC_setSocSampleWindow(myAdc, ADC_SocNumber_0, ADC_SocSampleWindow_7_cycles);  //seleciona a sample windows de SOC0 para 7 ciclos de clock, (6 ACQPS + 1)
    ADC_setSocSampleWindow(myAdc, ADC_SocNumber_1, ADC_SocSampleWindow_7_cycles);  //seleciona a sample windows de SOC1 para 7 ciclos de clock, (6 ACQPS + 1)
    ADC_setSocSampleWindow(myAdc, ADC_SocNumber_2, ADC_SocSampleWindow_7_cycles);  //seleciona a sample windows de SOC2 para 7 ciclos de clock, (6 ACQPS + 1)
```

Em seguida, configuramos o timer, configuramos as saídas GPIO e iniciamos a contagem do timer.

```
    // Configuração do cpu_timer0
    CLK_enableCpuTimerClock(myClk,CLK_CpuTimerNumber_0); // Habilita o Clock de cpu-timer0
    TIMER_stop(myTimer0); // Para a contagem do timer
    TIMER_setPeriod(myTimer0, period*60); // Atribui ao timer o período definido no escopo
    TIMER_setPreScaler(myTimer0, 0); // Configura o "prescale" do timer. O clock do timer é dividido pelo valor inserido mais uma unidade. Neste caso, será dividido por 1
    TIMER_reload(myTimer0); // Reinicia o timer
    TIMER_setEmulationMode(myTimer0, TIMER_EmulationMode_StopAfterNextDecrement); // Utilizado na emulação. Define, por exemplo, a ação do timer quando um "breakpoint" é inserido no software de desenvolvimento
    TIMER_enableInt(myTimer0); // Habilita a interrupção do timer

    Gpio_setup(); // Chama a função de configuração da GPIO

    TIMER_start(myTimer0); // Inicia a contagem do timer

    // Aguarda uma interrupção
    for(;;);
}
```

Implementamos as rotinas de interrupção do timer e do ADC.

```
// Rotina de interrupção do timer
__interrupt void cpu_timer0_isr(void)
{
    // Alterna o nível lógico das saídas dos LEDs
    GPIO_toggle(myGpio, GPIO_Number_2);
    GPIO_toggle(myGpio, GPIO_Number_3);

    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_1);
}

// Rotina de interrupção do ADC
__interrupt void adc_isr(void)
{
    //Devemos descartar o resultado da primeira conversão (SOC0) devido ao workaround de [rev0 silicon errata]:

    Voltage6 = ADC_readResult(myAdc, ADC_ResultNumber_1); // Recupera o valor da conversão do canal ADCINA4, associado à SOC1
```

```
    Voltage8 = ADC_readResult(myAdc, ADC_ResultNumber_2); // Recupera o valor da conversão
    do canal ADCINA2, associado à SOC2

    if(Voltage6 < Voltage8) // Compara os valores das tensões
    {
        // Acende o LED1 se a tensão no pino 8 é superior à do pino 6
        GPIO_setHigh(myGpio, GPIO_Number_0);
        GPIO_setLow(myGpio, GPIO_Number_1);
    }
    else
    {
        // Acende o LED0 caso contrário
        GPIO_setLow(myGpio, GPIO_Number_0);
        GPIO_setHigh(myGpio, GPIO_Number_1);
    }

    // Limpa o flag de ADCINT1 para o próximo evento
    ADC_clearIntFlag(myAdc, ADC_IntNumber_1);

    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_10);
}
```

O código condensado encontra-se no Anexo 2.

4. MÓDULO EPWM

4.1. VISÃO GERAL

Um periférico *PWM* é um circuito que gera formas de onda retangulares com frequência, fase e ciclo de trabalho bem definidos, sendo utilizado para diversos fins, como para o controle de máquinas elétricas, chaveamento de fontes, conversores e *UPS*, recuperação de sinais analógicos, etc. No microcontrolador TMS320F28027 podemos utilizá-lo, entre outras coisas, como trigger dos conversores analógicos digitais.

O F28027 nos fornece ao todo 4 módulos *ePWM* (*Enhanced Pulse Width Modulator*). Cada módulo possui duas saídas ePWM (*EPWMxA* e *EPWMxB*) que podem ser usadas como: duas saídas independentes com operação em borda simples; duas saídas independentes com borda dupla simétrica; uma saída independente com borda dupla assimétrica. Alguns dos módulos apresentam uma extensão chamada *HRPWM* (*High-Resolution PWM*) que permite um controle mais preciso e é utilizado em aplicações específicas como conversão Digital/Analógica.

Não abordaremos o HRPWM, mas o leitor interessado pode consultar o documento [8].

Um módulo ePWM é dividido em 8 submódulos:

- *Time-base* (TB);
- *Counter-compare* (CC);
- *Action-qualifier* (AQ);
- *Dead-band* (DB);
- *PWM-chopper* (PC);
- *Event-trigger* (ET);
- *Trip-zone* (TZ);
- *Digital Compare* (DC).

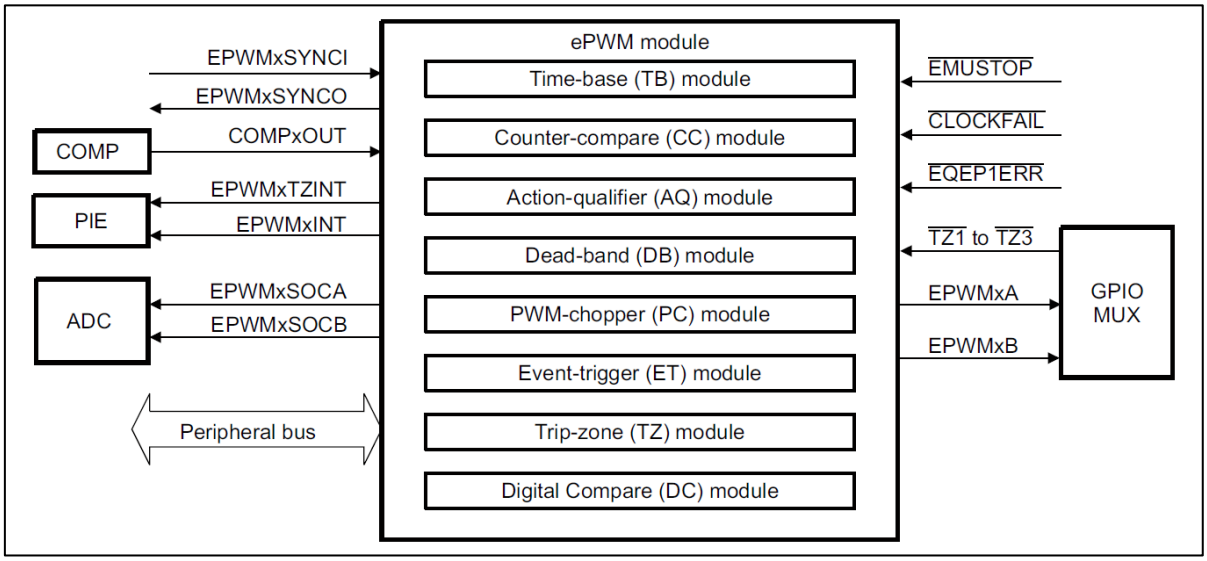


Figura 4.1: Diagrama simplificado dos submódulos ePWM.

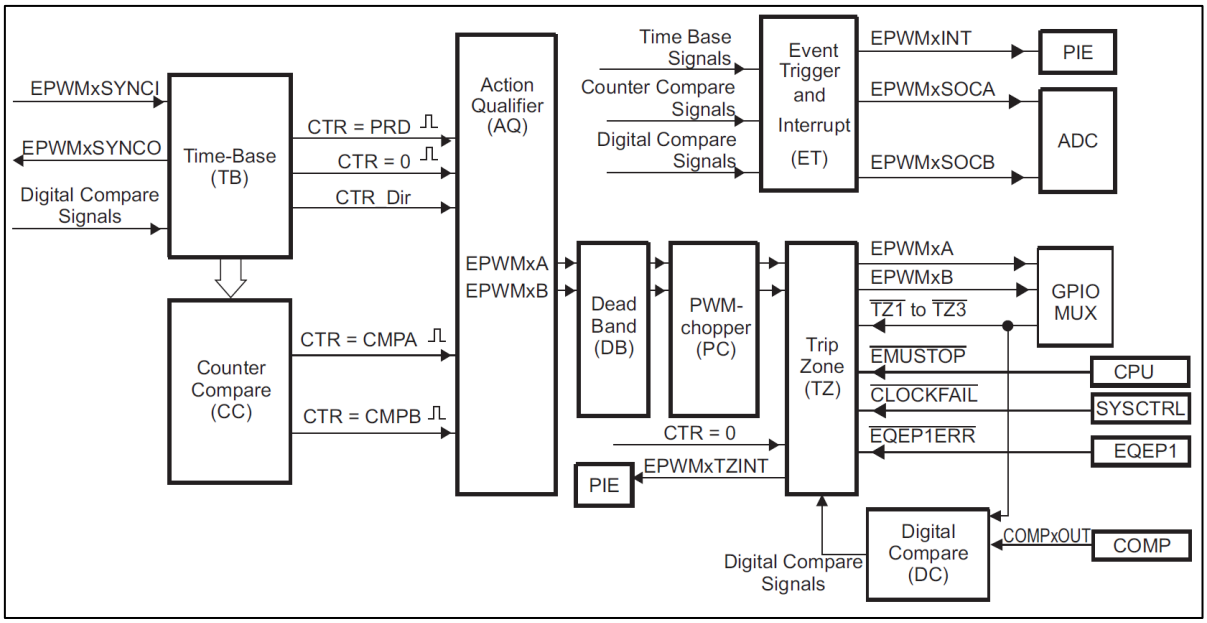


Figura 4.2: Diagrama em blocos de um módulo ePWM.

Abordaremos cada submódulo separadamente, dando enfoque ao exemplo que será explorado posteriormente. Portanto, não trabalharemos a fundo todos os blocos e recursos disponíveis no periférico, mas somente aqueles que permitam configurações e aplicações básicas. Cabe ao leitor consultar o documento [7] afim de obter mais informações.

Antes de passarmos aos submódulos, devemos propor algumas definições.

Active Register

Um registrador de controle do tipo *active* ou *ativo* se trata do registrador convencional: tem a função de controlar o hardware a partir das configurações armazenadas em seus bits. Em alguns registradores de determinados periféricos, o Piccolo oferece, paralelamente ao active, uma espécie de registrador virtual, chamado *Shadow Register*.

Shadow Register

Um registrador *shadow* ou *shadowed* funciona como um registrador temporário, cujas informações são repassadas ao registrador ativo em um momento específico. Ele não atua diretamente no controle do hardware, sendo utilizado para

evitar espúrios enquanto os bits do registrador estão sendo modificados via software. Em outras palavras, os periféricos são configurados de uma só vez ao se copiar o conteúdo do registrador shadowed ao registrador ativo.

Alguns registradores com esta opção apresentam seus shadow registers habilitados por padrão, mas há a opção de desabilitá-los.

4.2. TIME-BASE (TB)

Este submódulo, como o próprio nome já diz, é o que controla toda a base de tempo e sincronização do seu respectivo módulo ePWM. Entre suas funções, temos:

- Especificar o período (PRD) do contador de 16 bits (*time-base counter* – TBCTR);
- Gerir a sincronizar em relação a outros módulos ePWM;
- Gerir a relação de fase com os outros módulos;
- Configurar o tipo de contagem: progressiva, regressiva, ou progressiva e regressiva;
- Configurar o clock (TBCLK), podendo utilizar uma fração (PRESCALE) da frequência clock da cpu (SYSCLKOUT);
- Gerar pulsos de controle (CTR) quando o contador é igual a zero (CTR=0) ou igual ao período (CTR=PRD).

O registrador do período (TBPRD) possui shadow register inicialmente habilitado (TBCTL[PRDLD] = 0). O conteúdo do shadow register é transferido ao registrador ativo quando o contador é igual a zero (TBCTR=0). Se TBCTL[PRDLD] = 1, a escrita e leitura de TBPRD é direta ao registrador ativo.

O diagrama em blocos do Time-Base é mostrado abaixo a título de ilustração.

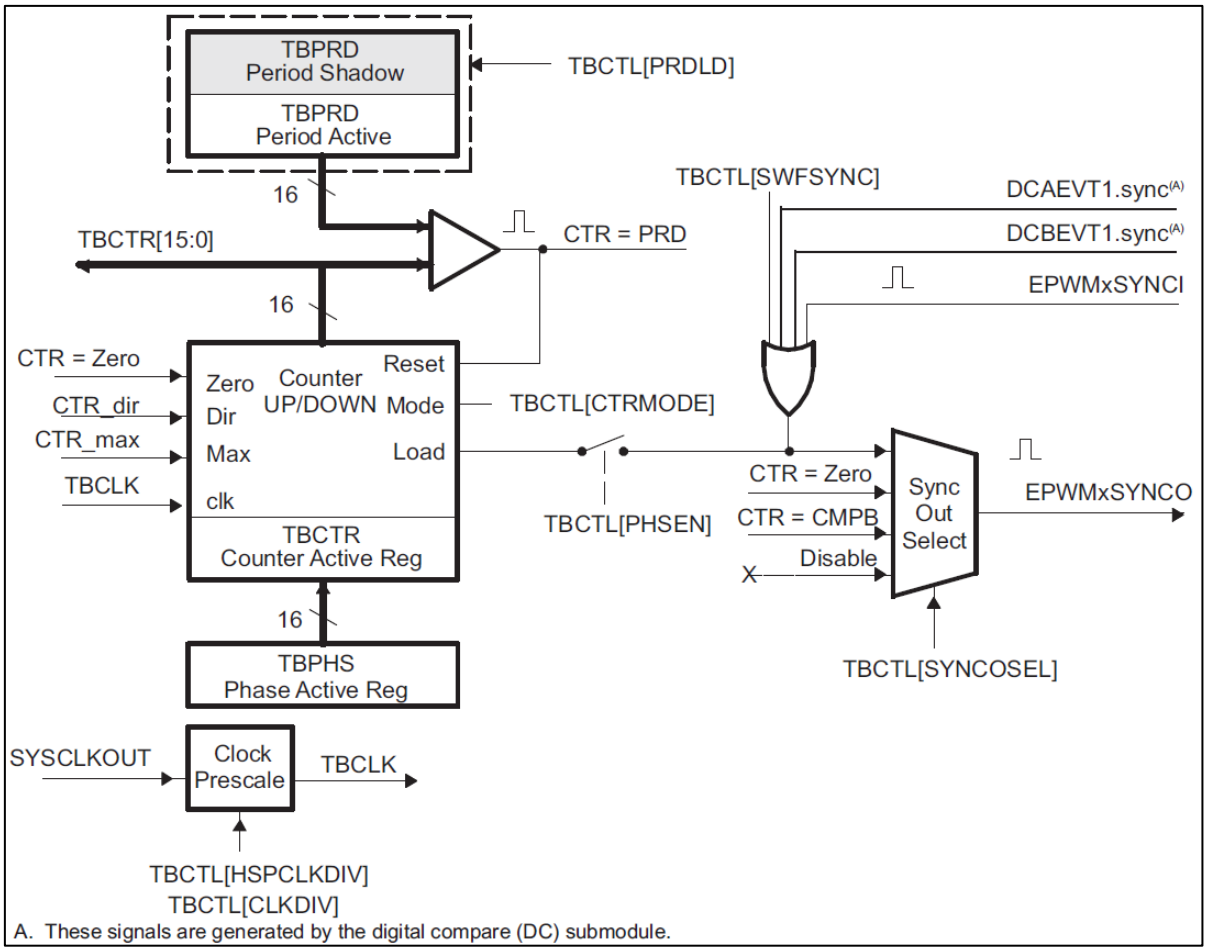


Figura 4.3: Diagrama em blocos do Time-Base.

Os valores de período (em segundos) e da frequência (em hertz) podem ser calculados de acordo com o tipo de contagem configurada:

a) Contagem progressiva: o contador inicia em zero, é incrementado até atingir o valor do período (PRD) e em seguida é zerado.

$$T = TBCLK * (PRD + 1) \quad , \quad f = \frac{1}{T} \tag{4.1}$$

b) Contagem regressiva: o contador inicia em PRD, é decrementado até atingir zero e em seguida é carregado com PRD novamente.

$$T = TBCLK * (PRD + 1) \quad , \quad f = \frac{1}{T} \tag{4.2}$$

c) Contagem progressiva e regressiva: o contador inicia em zero, é incrementado até PRD e em seguida é decrementado até atingir zero novamente.

$$T = 2 * TBCLK * PRD \quad , \quad f = \frac{1}{T} \tag{4.3}$$

Cada módulo ePWM possui um bit de sincronização de entrada (EPWMxSYNCI) e um de saída (EPWMxSYNCO) que são interconectados de determinada maneira, de forma a possibilitar as sincronizações de eventos e fases entre diferentes módulos. A explicação detalhada deste e outros assuntos encontra-se no documento [7].

4.3. COUNTER-COMPARE (CC)

O Counter-Compare é o submódulo que tem a função de gerar os pulsos do ePWM através da comparação contínua entre o valor atual do contador TBCTR e o valor de referência definido pelo usuário. Desta forma, ele é capaz de controlar o ciclo de trabalho da PWM.

Ele possui os registradores CMPA e CMPB, que armazenam os valores de referência. Ambos os registradores possuem shadow register, os quais estão inicialmente habilitados (CMPCLT[SHDWAMODE] = CMPCLT[SHDWBMODE] = 0). Para desabilitar o shadow mode, fazemos CMPCLT[SHDWxMODE] = 1.

Através dos dois comparadores, podemos gerar formas de ondas simétricas e assimétricas:

- Contagem progressiva: gera uma forma de onda assimétrica;
- Contagem regressiva: gera uma forma de onda assimétrica;
- Contagem progressiva e regressiva: gera uma forma de onda simétrica.

As figuras abaixo, retiradas de [7], apresentam alguns exemplos de formas de ondas que podem ser geradas.

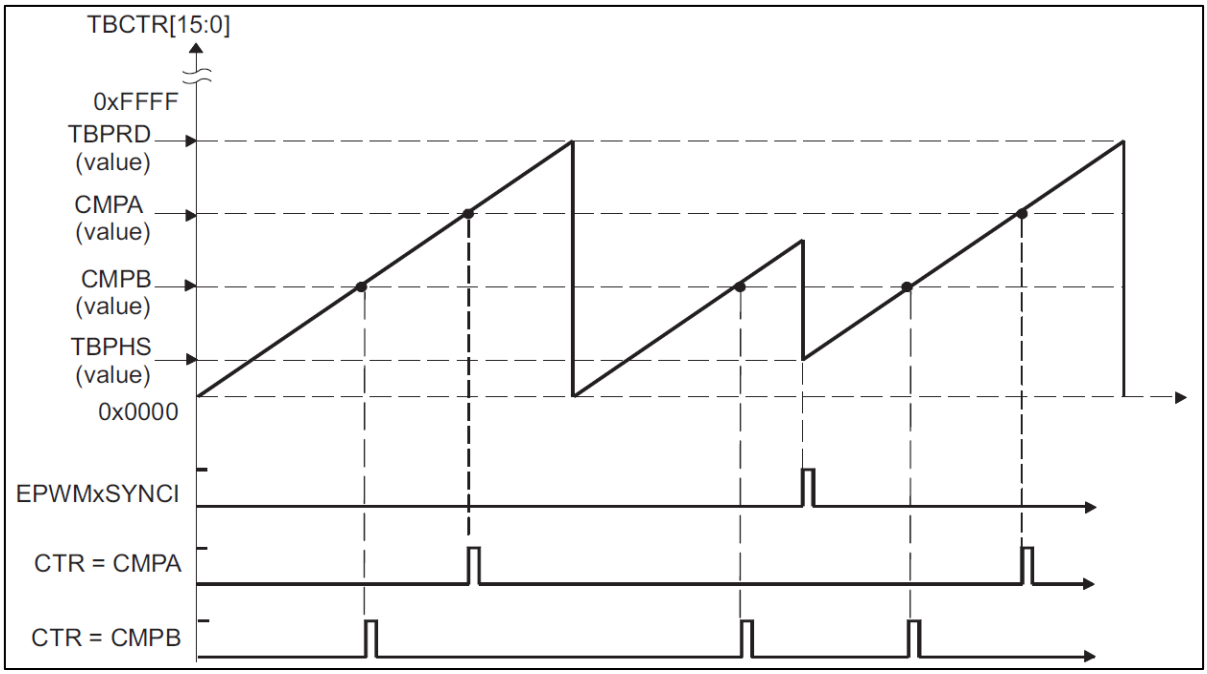


Figura 4.4: Formas de onda geradas em contagem progressiva.

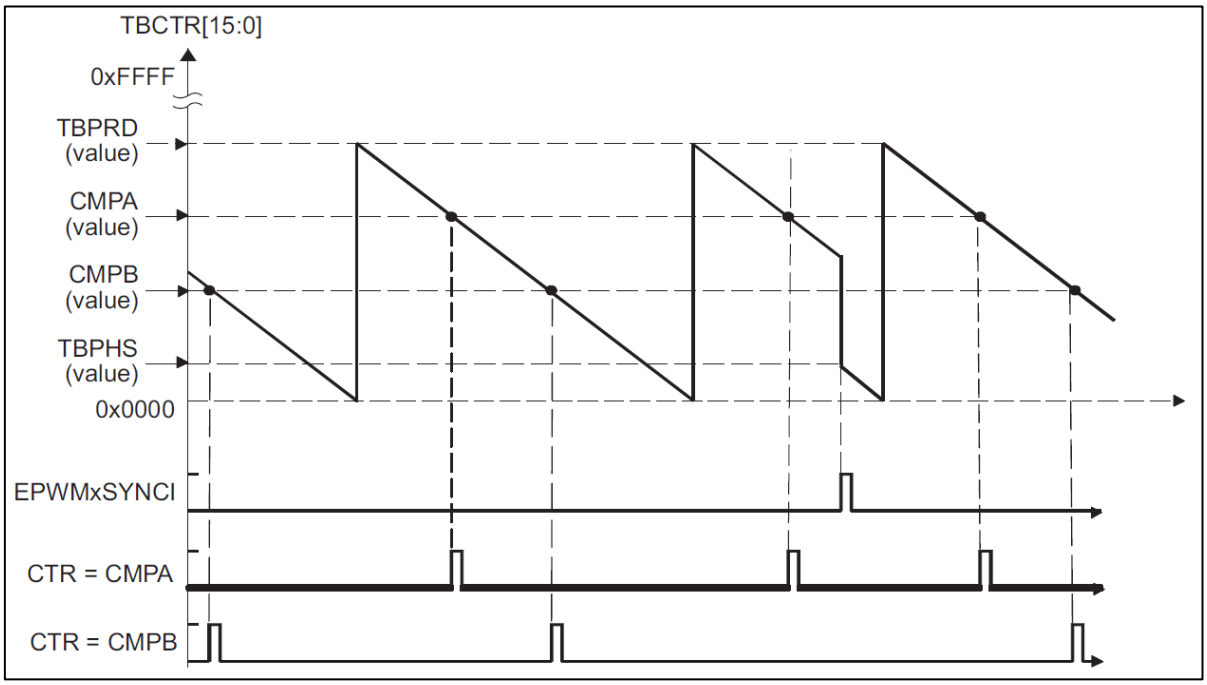


Figura 4.5: Formas de onda geradas em contagem regressiva.

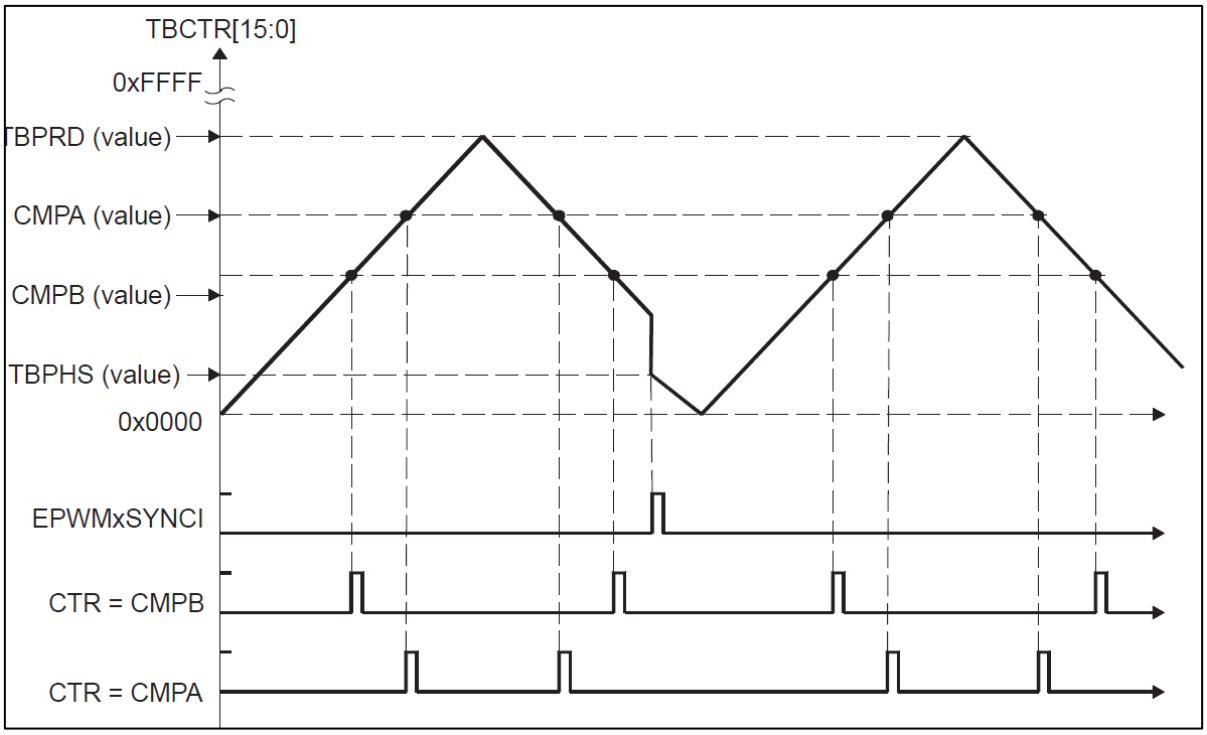


Figura 4.6: Formas de onda geradas em contagem progressiva e regressiva.

4.4. ACTION-QUALIFIER (AQ)

Este bloco é o responsável por contruir as formas de onda de saída do ePWM. Ele recebe os pulsos gerados por TB quando CTR=PRD e CTR=0, e os provenientes de CC quando CTR=CMPA e CTR=CMPB. Através destes eventos, ele define a ação a ser executada, sejam elas: levar as saídas EPWMxA ou EPWMxB a nível lógico alto (SET); levar as saídas a nível lógico baixo (CLEAR); inverter o nível lógico, o que equivale a uma operação “not” (TOGGLE); manter o nível lógico atual das saídas (em outras palavras, não fazer nada). Esta última ação, apesar de não alterar o sinal de saída, tem utilidade quando deseja-se utilizar o ePWM como trigger do ADC ou disparar uma interrupção na CPU.

É importante mencionar que as saídas EPWMxA e EPWMxB podem ser utilizadas independentemente. Além disso, uma única saída pode utilizar todos os pulsos (CTR=PRD, CTR=0, CTR=CMPA e CTR=CMPB) para gerar sua forma de onda.

Temos ainda um quinto evento que gera uma ação no Action-Qualifier, que é o evento disparado por software. Seu registrador associado (AQCSFRC – Action-Qualifier Continuous Software Force) é o único que contém shadow register.

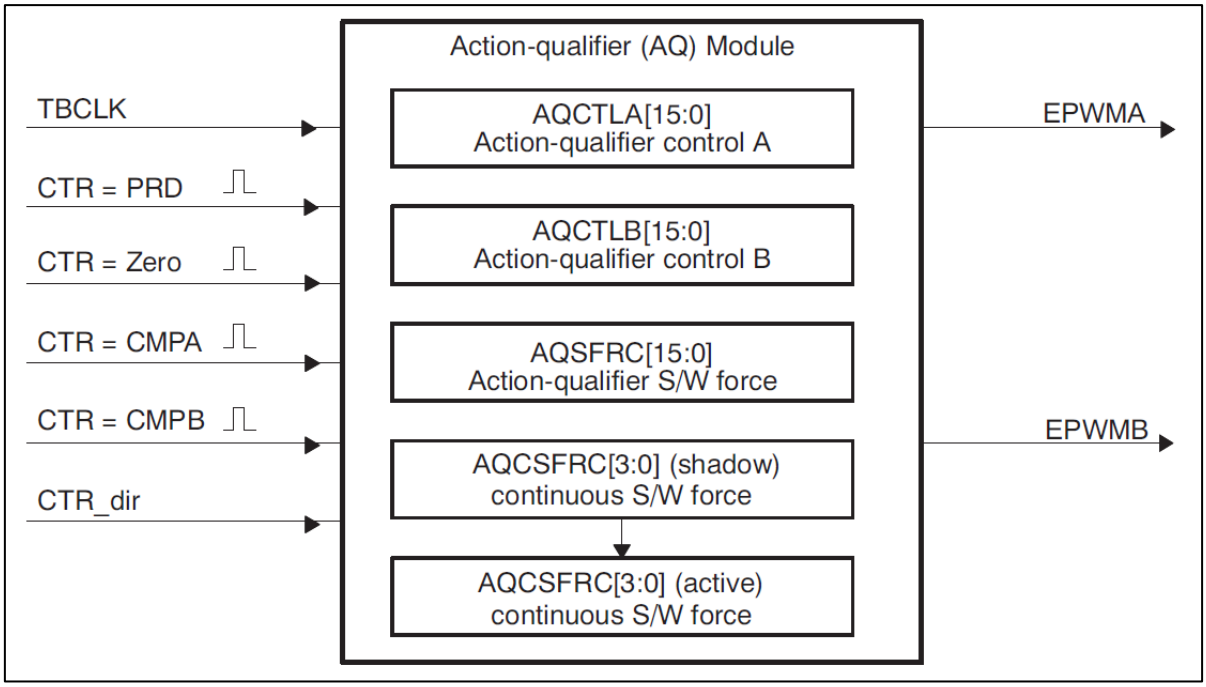


Figura 4.7: Diagrama de entradas e saídas do Action-Qualifier.

4.5. DEAD-BAND (DB)

O submódulo Dead-Band é um recurso interessante quando deseja-se gerar um par de sinais relacionados. Desta forma, podemos gerar o par de sinais EPWMxA e EPWMxB relacionados a partir unicamente da configuração da saída EPWMxA, e, utilizando o DB, construímos o sinal de EPWMxB. Temos as seguintes opções:

- *Active high* (AH) e *Active low* (AL): EPWMxB acompanha o nível lógico de EPWMxA;
- *Active high complementary* (AHC) e *Active low complementary* (AHC): EPWMxB acompanha o nível lógico de EPWMxA de forma complementar, isto é, se EPWMxA sofre uma transição de subida, EPWMxB sofre uma transição de descida e vice-versa;
- *Rising edge delay* (RED) *e Falling edge delay* (FED): uma saída sofre um atraso em relação à outra na borda de subida, de descida, ou ambas.

Notem pelo diagrama em blocos, o qual é reproduzido abaixo, que os sinais de saída passam obrigatoriamente pelo módulo DB. Portanto, ele também apresenta a opção de não executar nenhuma ação nos sinais (bypass).

Não entraremos em mais detalhes sobre este bloco. As demais informações podem ser encontradas em [7].

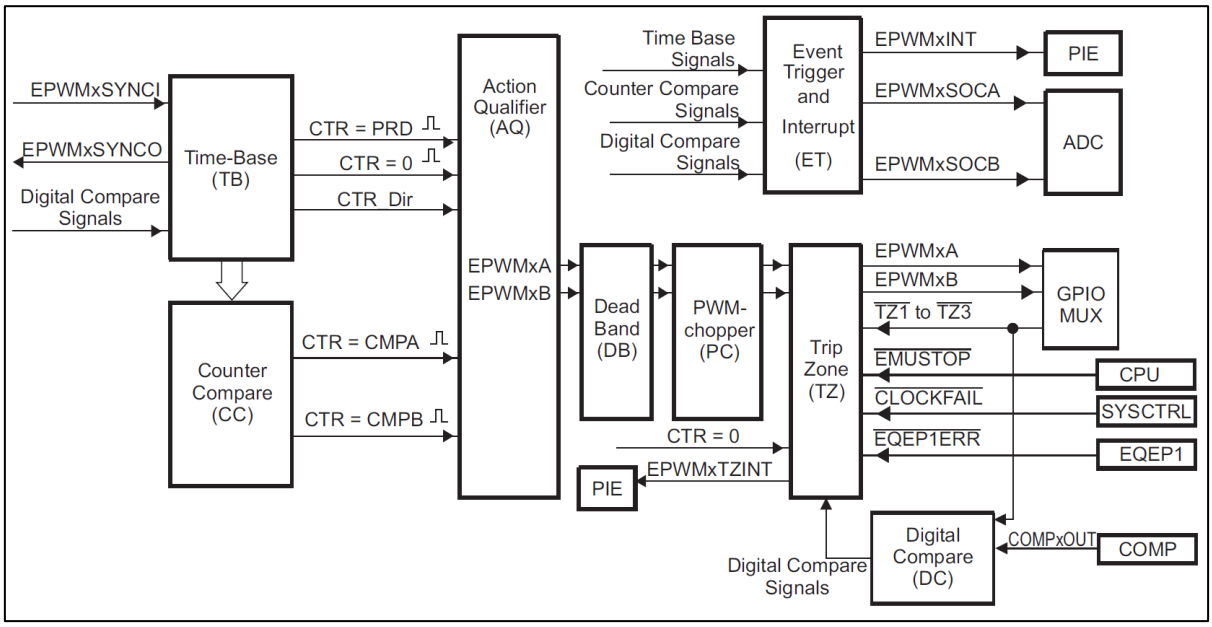


Figura 4.8: Diagrama em blocos de um módulo ePWM.

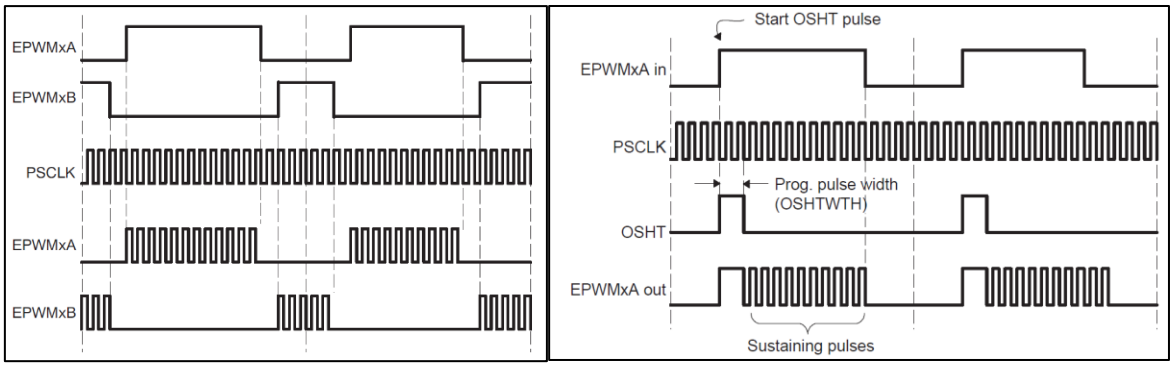
4.6. PWM-CHOPPER (PC)

No diagrama em blocos anterior, podemos ver que o bloco PWM-Chopper recebe os sinais de saída provenientes do bloco Dead-Band. Ele tem a função de modular estes sinais com uma portadora de alta frequência. Este recurso é útil em algumas aplicações em Eletrônica de Potência, por exemplo, no controle de chaves eletrônicas com transformadores de pulso.

O submódulo PC é capaz de:

- Programar a frequência da portadora;
- Programar a largura dos pulsos ou do primeiro pulso;
- Programar o ciclo de trabalho do segundo pulso em diante;
- Não realizar nenhuma operação com os sinais (bypass).

Exemplos de formas de onda que podem ser geradas estão mostradas na figura 4.9. Não abordaremos a fundo este bloco.



(a) (b)

Figura 4.9: Formas de onda moduladas com o submódulo PWM-Chopper: (a) Modulação simples; (b) Modulação com primeiro pulso (ONESHOT pulse).

4.7. TRIP-ZONE (TZ)

O submódulo Trip-Zone tem a finalidade de tratar faltas. Ao todo, existem 6 sinais *trip-zone* (TZ1 a TZ6): os sinais TZ1 a TZ3 são conectados à GPIO e podem ser configuradas como entradas assíncronas; TZ4 é conectado ao sinal de erro EQEP1ERR proveniente do módulo EQEP[1]; TZ5 é conectado ao sistema de controle

[1] O F28027 não possui o módulo EQEP (Enhanced Quadrature Encoder Pulse), portanto este sinal trip-zone não está disponível. Ver documento [9] para mais informações sobre o módulo, e [10] para ver quais dispositivos possuem este periférico.

(CLOCKFAIL[2]); TZ6 é conectada ao bit EMUSTOP[3] da CPU. Cada módulo ePWM pode ser configurado para usar ou ignorar um ou mais destes sinais.

Os sinais trip-zones são os que indicam a presença de falhas no sistema. O submódulo TZ é capaz de reagir a estes sinais e a forçar as saídas EPWMxA e EPWMxB a uma das condições abaixo:

- Nível lógico alto;
- Nível lógico baixo;
- Alta impedância;
- Nenhuma ação.

Maiores informações estão disponíveis em [7].

4.8. EVENT-TRIGGER (ET)

O Event-Trigger é um bloco importante quando um módulo ePWM é utilizado para gerar interrupções na CPU ou como trigger do ADC. Entre suas principais funções: receber os eventos gerados pelos módulos Time-Base, Counter-Compare e Digital Compare; gerar interrupções na CPU ou disparar o ADC para todo evento, cada segundo evento ou cada terceiro evento.

Cada módulo ePWM está conectado à PIE por meio de um bit EPWMxINT e ao ADC por meio dos bits EPWMxSOCA ou EPWMxSOCB. A figura abaixo esquematiza essas conexões.

[2] Este bit é proveniente do sistema de controle (SYSCTRL). Quando o usuário seleciona a fonte de clock External Oscillator ou Internal Oscillator 2, e é detectada a falta de pulsos de clock, o bit CLOCKFAIL é ativado e a fonte de clock é automaticamente comutada para Internal Oscillator 1. Ver documento [5].

[3] Há poucas informações sobre este bit na documentação do dispositivo. Ele é ativado quando há um bug/interrupção/suspensão da CPU, normalmente quando é conectado um emulador, ou sessão de emulação ou debug no dispositivo. Ver [5].

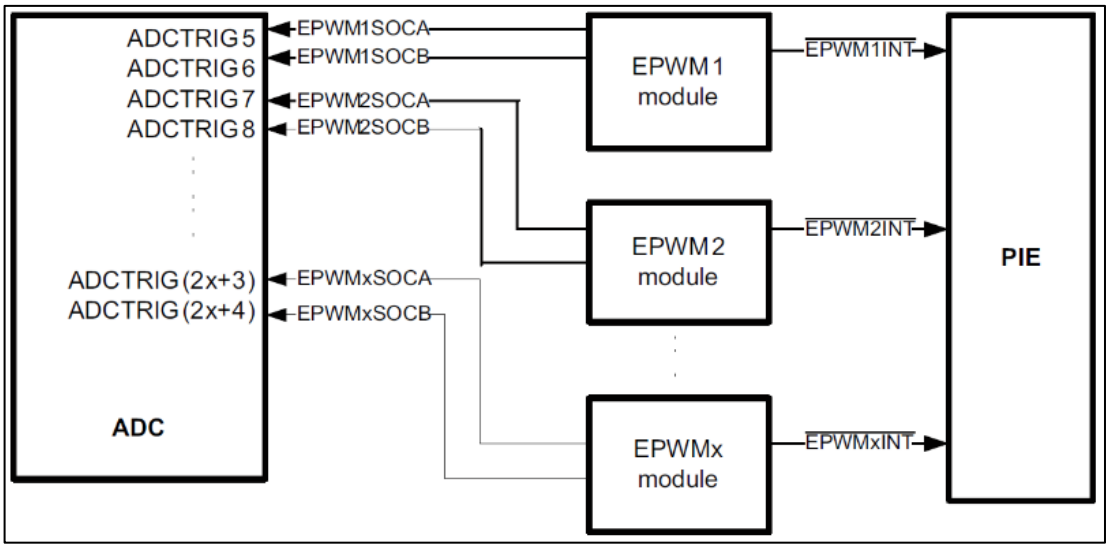


Figura 4.10: Conexões dos bits do módulo Event-trigger.

O submódulo é configurado para gerar um pulso, dado um tipo de evento entre os listados abaixo:

- TBCTR = 0;
- TBCTR = PRD;
- TBCTR = 0 OU TBCTR = PRD;
- TBCTR = CMPA quando o contador está sendo incrementado;
- TBCTR = CMPA quando o contador está sendo decrementado;
- TBCTR = CMPB quando o contador está sendo incrementado;
- TBCTR = CMPA quando o contador está sendo decrementado;

4.9. DIGITAL COMPARE (DC)

O último bloco trata-se do Digital Compare. Como mostra o diagrama em blocos da figura 4.11, ele recebe sinais externos à PWM provenientes do Comparador Analógico (COMPxOUT) (ver [4]) e os sinais TZ1 a TZ3 da GPIO. Suas saídas são entradas dos blocos Trip-Zone, Event-Trigger e Time-Base. Sua principal função é gerar eventos que são utilizados nesses submódulos.

Os bits DCAH (Digital Compare A High), DCAL (Digital Compare A Low) , DCBH (Digital Compare B High) e DCBL (Digital Compare B Low) são obtidos à partir de lógica digital com as entradas do submódulo e podem ser filtrados ou utilizados diretamente para: gerar interrupções trip-zone; disparar conversões no ADC; forçar um evento; gerar pulsos de sincronização entre módulos ePWM.

Ele pode ainda ser utilizado para filtrar eventos para remoção de ruídos, principalmente originados do Comparador Analógico. A figura abaixo esquematiza o funcionamento do Digital Compare. Os demais detalhes são apresentados em [7].

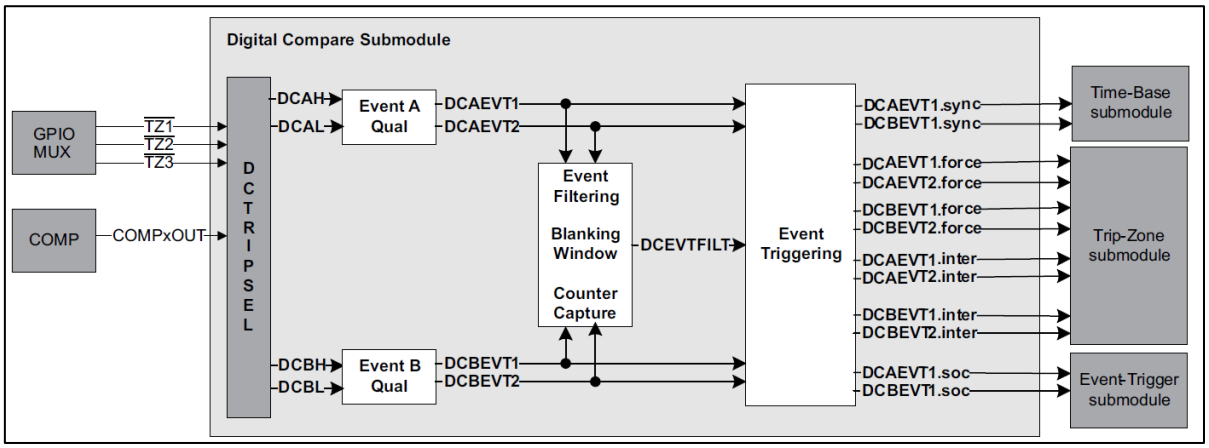


Figura 4.11: Diagrama em blocos do Digital Compare.

4.10. EXEMPLO DE CÓDIGO UTILIZANDO O EPWM

EXEMPLO 2 – CONTROLE DE BRILHO DO LED COM POTÊNCIOMETRO

No exemplo a seguir, utilizaremos um módulo ePWM para modificar o brilho de dois LEDs do launchpad, de acordo com o valor da tensão no pino 8 do

barramento J1, que está conectado à entrada ADCINA2 do conversor A/D. O módulo EPWM1A será utilizado tanto para o controle dos LEDs quanto para disparar as conversões do ADC.

O início do código segue o padrão do exemplo 1:

```
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo

#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog
```

Incluímos uma definição para o período do time-base de ePWM1. Declaramos as funções que implementaremos, a rotina de interrupção que o ADC executará, bem como a variável global *Voltage* que armazenará o valor da conversão. Declaramos os handles globais.

```
#define PWM1_TIMER_TBPRD   0xFFFF // Define o período do timer de ePWM1

void InitEPwmTimer(void); // Declara a rotina de configuração da ePWM
void InitAdc(void); // Declara a rotina de configuração do ADC
void InitGpio(void); // Declara a rotina de configuração do GPIO

__interrupt void adc_isr(void); //Protótipo da rotina de interrupção

uint16_t Voltage; // Variável que armazena o valor da tensão obtido pelo ADC

// Declara os handles globais
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm1;
ADC_Handle myAdc;
CPU_Handle myCpu;
```

Na sequência, mostramos a função principal completa. Notem que as configurações são as mesmas do exemplo 1.

```
int main(void)
{
    // Declara handles locais
    PLL_Handle myPll;
```

```
    WDOG_Handle myWDog;

    // Inicializa os handles
    myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
    myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
    myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
    myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
    myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
    myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
    myPwm1 = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
    myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));
    myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));

    // Executa a inicialização básica do sistema:
    WDOG_disable(myWDog);
    CLK_enableAdcClock(myClk); // Habilita o clock do ADC
    (*Device_cal)();

    //Seleciona o oscilador interno 1 para ser o clock:
    CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

    // Configura a PLL para x12 / 2 que resulta em  60Mhz = 10Mhz * 12 / 2:
    PLL_setup(myPll, PLL_Multiplier_12, PLL_DivideSelect_ClkIn_by_2);

    // Desabilita a PIE e todas as interrupções, e limpa os flags:
    PIE_disable(myPie);
    PIE_disableAllInts(myPie);
    CPU_disableGlobalInts(myCpu);
    CPU_clearIntFlags(myCpu);

    // [Usado no modo de configuração "FLASH" do CC] Copia funções da RAM para a RAM:
    #ifdef _FLASH
      memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
    #endif

    // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
    PIE_setDebugIntVectorTable(myPie); // *Somente no modo de configuração DEBUG
    PIE_enable(myPie);

    // Configura a rotina de interrupção do ADCINT1 na PIE:
    PIE_registerPieIntHandler(myPie, PIE_GroupNumber_10, PIE_SubGroupNumber_1,
                    (intVec_t)&adc_isr);

    // Chamam as rotinas de configuração do ePWM, ADC e GPIO, respectivamente. Elas serão
    implementadas posteriormente
    InitEPwmTimer();
    InitAdc();
    InitGpio();

    CPU_enableGlobalInts(myCpu); // Habilita a interrupção global INTM (ver ** control interrupt)

    CPU_enableDebugInt(myCpu); // Habilita a interrupção global do modo de emulação em tempo
    real DBGM (ver ** control interrupt)

    for(;;); // Aguarda as interrupções
}
```

Implementaremos a função que configura o módulo ePWM. Primeiramente, desabilitamos o clock dos TBs de todos os módulos e, em seguida, habilitamos o clock de ePWM1. O clock do time-base de ePWM1 permanece desabilitado, e será ativado após todas as configurações do bloco.

```
// Rotina de configuração do ePWM
void InitEPwmTimer(void)
{
    CLK_disableTbClockSync(myClk); // Desabilita o clock de todos os TBs

    CLK_enablePwmClock(myClk, PWM_Number_1); // Habilita o clock de ePWM1
```

Configuramos o período e o modo de contagem (progressivo e regressivo). Definimos o valor a ser comparado pelo comparador do CC e as ações a serem realizadas a cada evento.

```
    // Configurações do bloco TB
    PWM_setPeriod(myPwm1, PWM1_TIMER_TBPRD); //Define o período com base na definição no escopo

    PWM_setCounterMode(myPwm1, PWM_CounterMode_UpDown); // Define contagem prog. e regres.

    // Configurações do bloco CC
    PWM_setCmpA(myPwm1, 0); // Define o valor 0 a ser comparado com o contador. Este valor será posteriormente substituído pela tensão medida

    // Configurações do bloco AQ; Notem que as saídas EPWM1A e EPWM1B foram configuradas para serem complementares. Poderíamos utilizar o bloco Dead-Band para configurar saídas complementares.
    PWM_setActionQual_CntUp_CmpA_PwmA(myPwm1,PWM_ActionQual_Clear); // Leva o nível lógico de EPWM1A a 0 quando, na contagem progressiva, o valor de referência for igual ao contador

    PWM_setActionQual_CntDown_CmpA_PwmA(myPwm1,PWM_ActionQual_Set); // Leva o nível lógico de EPWM1A a 1 quando, na contagem regressiva, o valor de referência for igual ao contador

    PWM_setActionQual_CntUp_CmpA_PwmB(myPwm1,PWM_ActionQual_Set); // Leva o nível lógico de EPWM1B a 1 quando, na contagem progressiva, o valor de referência for igual ao contador

    PWM_setActionQual_CntDown_CmpA_PwmB(myPwm1,PWM_ActionQual_Clear); // Leva o nível lógico de EPWM1A a 0 quando, na contagem regressiva, o valor de referência for igual ao contador
```

As configurações a seguir são utilizadas somente no caso em que se utiliza algum ePWM como trigger em entradas do ADC e são as mesmas do exemplo anterior.

```
    PWM_enableSocAPulse(myPwm1); // Habilita a geração de pulsos para SOCs (grupo A)

    PWM_setSocAPulseSrc(myPwm1, PWM_SocPulseSrc_CounterEqualPeriod); // Seleciona a
    condição para geração do pulso. Nesse caso, o disparo ocorrerá quando o valor do contador da
    PWM for igual ao período

    PWM_setSocAPeriod(myPwm1, PWM_SocPeriod_FirstEvent); // Define que o pulso será
    gerado no primeiro evento
```

Por fim, reabilitamos os clocks dos TBs:

```
    CLK_enableTbClockSync(myClk); // Habilita clock de todos os TB
}
```

Implementaremos agora a função que configura o ADC. Como a configuração é exatamente a mesma, não entraremos novamente em detalhes. A única diferença em relação ao exemplo 1 é que utilizamos somente a entrada ADCINA2, que foi associada a dois SOCs de forma a realizar o workaround provido em [6].

```
// Configuração do ADC
void InitAdc(void)
{
    // Inicialização básica do ADC:
    ADC_enableBandGap(myAdc); // Habilita o circuito da tensão de referência
    ADC_enableRefBuffers(myAdc); // Habilita o buffer do circuito de referência
    ADC_powerUp(myAdc); // Energiza os circuitos do ADC (ver tópico Power Up Sequence em **)
    ADC_enable(myAdc); // Habilita o ADC (ver tópico Power Up Sequence em **)
    ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int); // Seleciona a referência interna

    PIE_enableAdcInt(myPie, ADC_IntNumber_1); // Habilita ADCINT1 na PIE

    CPU_enableInt(myCpu, CPU_IntNumber_10); // Habilita CPU Interrupt 10 (grupo 10 da PIE -
    PIE_GroupNumber_X -> CPU_IntNumber_X)

    // Configuração do ADC
    //OBS: O canal ADCINA2 será associado a dois SOCs para ser duplamente amostrado, como
    workaround provido no documento [rev0 silicon errata]

    ADC_setIntPulseGenMode(myAdc, ADC_IntPulseGenMode_Prior); //Configura o pulso da
    interrupção ADCINT1 para ocorrer no fim da conversão

    ADC_enableInt(myAdc, ADC_IntNumber_1); //Habilita a interrupção ADCINT1

    ADC_setIntMode(myAdc, ADC_IntNumber_1, ADC_IntMode_ClearFlag); //Seleciona ADCINT1
    para o modo Clear Flag (uma nova interrupção não pode ser gerada enquanto o flag não for
    resetado)

    ADC_setIntSrc(myAdc, ADC_IntNumber_1, ADC_IntSrc_EOC1); //Configura EOC1 (associado
    a SOC1) para disparar ADCINT1
```

```
    ADC_setSocChanNumber (myAdc, ADC_SocNumber_0, ADC_SocChanNumber_A2); //associa SOC0 no canal ADCINA2

    ADC_setSocChanNumber (myAdc, ADC_SocNumber_1, ADC_SocChanNumber_A2); //associa SOC1 no canal ADCINA2

    ADC_setSocTrigSrc(myAdc, ADC_SocNumber_0, ADC_SocTrigSrc_EPWM1_ADCSOCA); //seleciona EPWM1A para disparar SOC0 (devido ao round-robin, SOC0 é amostrado antes de SOC1

    ADC_setSocTrigSrc(myAdc, ADC_SocNumber_1, ADC_SocTrigSrc_EPWM1_ADCSOCA); //seleciona EPWM1A para disparar SOC1 (devido ao round-robin, SOC0 é amostrado antes de SOC1

    ADC_setSocSampleWindow(myAdc, ADC_SocNumber_0,
     ADC_SocSampleWindow_7_cycles); //seleciona a sample windows de SOC0 para 7 ciclos de clock, (6 ACQPS + 1)

    ADC_setSocSampleWindow(myAdc, ADC_SocNumber_1,
    ADC_SocSampleWindow_7_cycles); //seleciona a sample windows de SOC1 para 7 ciclos de clock, (6 ACQPS + 1)
}
```

A próxima rotina é a que configura o GPIO.

```
// Configura GPIO
void InitGpio(void)
{
    // Habilita resistores de pull-up
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Enable);

    // Define os pinos no modo ePWM, que recebe o nível lógico proveniente das saídas dos módulos EPWM1A e EPWM1B
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_EPWM1A);
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_EPWM1B);
}
```

Por fim, implementamos a rotina de interrupção do ADC, que será responsável por medir a tensão na entrada ADCINA2 e, com base no valor medido, alterar a referência do comparador de ePWM1, de forma a modificar o brilho dos LEDs. Como o ADC é de 12 bits, e o contador do módulo ePWM é de 16 bits, devemos fazer um deslocamento de 4 bits na tensão medida, de forma que os LEDs variem do brilho mínimo ao máximo.

```
// Rotina de interrupção do ADC
__interrupt void adc_isr(void)
{
    Voltage = ADC_readResult(myAdc, ADC_ResultNumber_1); // Recupera o valor da conversão de SOC1, armazenado em ADCRESULT1
```

```
    PWM_setCmpA(myPwm1, Voltage<<4); // Define a tensão de referência de CMPA, de forma a
    variar o ciclo de trabalho dass saías EPWM1A e EPWM1B

    // Limpa o flag de ADCINT1 para o próximo evento
    ADC_clearIntFlag(myAdc, ADC_IntNumber_1);

    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_10);
}
```

O código condensado se encontra no Anexo 2.

5 CONCLUSÃO

O presente trabalho teve como objetivo explorar os recursos do microcontrolador TMS320F28027 do Piccolo Launchpad C200 - kit de baixo custo produzido pela Texas Instruments. O kit permite aplicações em sistemas com controle em tempo real.

Apresentamos os passos iniciais para a instalação e importação de um projeto no software Code Composer Studio, versão 6.1.3. Ele trabalha juntamente com o ControlSuite, fornecendo uma série de exemplos e aplicações a serem executadas no Launchpad.

Abordamos as características do seu conversor Analógico/Digital e do seu periférico PWM, detalhando os blocos internos e as funcionalidades de cada um deles. Utilizamos o modelo de programação Software Driver, que abstrai o acesso aos registradores, facilitando a programação e o entendimento do código por parte do leitor. O modelo Direct Acces Register requer conhecimentos de cada registrador e bit dos periféricos do microcontrolador, que estão disponíveis nos manuais do dispositivo indicados nas referências. Em aplicações em que não há a necessidade de tempo de execução ótimo, o modelo SD é suficiente para um sistema com bom desempenho.

Esperamos que o que foi apresentado até aqui sirva como ponto inicial para o desenvolvimento de protótipos com Launchpad. Muito do que foi apresentado se aplica a outros microcontroladores e kits de desenvolvimento produzidos pela Texas Instruments.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Code Composer Studio:

http://www.ti.com/tool/ccstudio

[2] ControlSuite:

http://www.ti.com/tool/controlsuite

[3] SPRUHX9 - F2802x Peripheral Driver Library:

http://www.ti.com/lit/ug/spruhx9/spruhx9.pdf

[4] SPRUGE5F - Piccolo Analog-to-Digital Converter (ADC) and Comparator:

http://www.ti.com/lit/ug/spruge5f/spruge5f.pdf

[5] SPRUFN3D - Piccolo System Control and Interrupts:

http://www.ti.com/lit/ug/sprufn3d/sprufn3d.pdf

[6] SPRZ292M - Silicon Errata:

http://www.ti.com/lit/er/sprz292m/sprz292m.pdf

[7] SPRUGE9E - Enhanced Pulse Width Modulator (ePWM) Module:

http://www.ti.com/lit/ug/spruge9e/spruge9e.pdf

[8] SPRUGE8E - Piccolo High Resolution Pulse Width Modulator (HRPWM):

http://www.ti.com/lit/ug/spruge8e/spruge8e.pdf

[9] SPRU790D - Enhanced Quadrature Encoder Pulse (eQEP) Module:

http://www.ti.com/lit/ug/spru790d/spru790d.pdf

[10] SPRU566L - C2000 Real-Time Control Peripherals:

http://www.ti.com/lit/ug/spru566l/spru566l.pdf

ANEXO 1

```
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo

#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog


__interrupt void adc_isr(void); //Protótipo da rotina de interrupção

// Variáveis globais utilizadas nesse exemplo:
uint16_t LoopCount;
uint16_t Voltage6; //Tensão no pino 6
uint16_t Voltage8; //Tensão no pino 8


// Declara os handles globais:
ADC_Handle myAdc;
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm;


void Gpio_setup() // Configura as saídas conectadas aos LEDs no GPIO
{

    // Habilita resistores de pull-up:
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Enable);

    // Seta os pinos no modo “propósito geral”:
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_GeneralPurpose);

    // Define os pinos como saídas
    GPIO_setDirection(myGpio, GPIO_Number_0, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_1, GPIO_Direction_Output);

    // Seta as saídas em nível lógico baixo, o que apaga os LEDs:
    GPIO_setHigh(myGpio, GPIO_Number_0);
    GPIO_setHigh(myGpio, GPIO_Number_1);
}
void main(void)
{
     // Declara os handles locais:
    CPU_Handle myCpu;
    PLL_Handle myPll;
    WDOG_Handle myWDog;
```

```
// Inicializa todos os handles da aplicação:
myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));
myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
myPwm = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));


// Executa a inicialização básica do sistema:
WDOG_disable(myWDog);
CLK_enableAdcClock(myClk); // Habilita o clock do ADC
(*Device_cal)();

//Seleciona o oscilador interno 1 para ser o clock:
CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

// Configura a PLL para x12 / 2 que resulta em  60Mhz = 10Mhz * 12 / 2:
PLL_setup(myPll, PLL_Multiplier_12, PLL_DivideSelect_ClkIn_by_2);

// Desabilita a PIE e todas as interrupções, e limpa os flags:
PIE_disable(myPie);
PIE_disableAllInts(myPie);
CPU_disableGlobalInts(myCpu);
CPU_clearIntFlags(myCpu);

// [Usado no modo de configuração "FLASH" do CC] Copia funções da RAM para a RAM:
#ifdef _FLASH
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
#endif

// Configura a tabela de vetores do modo DEBUG e habilita a PIE:
PIE_setDebugIntVectorTable(myPie); // *Somente no modo de configuração DEBUG
PIE_enable(myPie);


// Configura a rotina de interrupção do ADCINT1 na PIE:
PIE_registerPieIntHandler(myPie, PIE_GroupNumber_10, PIE_SubGroupNumber_1,
(intVec_t)&adc_isr);


// Inicialização básica do ADC:
ADC_enableBandGap(myAdc); // Habilita o circuito da tensão de referência
ADC_enableRefBuffers(myAdc); // Habilita o buffer do circuito de referência
ADC_powerUp(myAdc); // Energiza os circuitos do ADC (ver tópico Power Up Sequence em [4])
ADC_enable(myAdc); // Habilita o ADC (ver tópico Power Up Sequence em [4])
ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int); // Seleciona a referência interna


PIE_enableAdcInt(myPie, ADC_IntNumber_1); // Habilita ADCINT1 na PIE

CPU_enableInt(myCpu, CPU_IntNumber_10); // Habilita CPU Interrupt 10 (grupo 10 da PIE -
PIE_GroupNumber_X -> CPU_IntNumber_X)

CPU_enableGlobalInts(myCpu); // Habilita a interrupção global INTM (ver [5])
```

CPU_enableDebugInt(myCpu); *// Habilita a interrupção global do modo de emulação em tempo real DBGM (ver [5])*

// Configuração do ADC
//OBS: O canal ADCINA4 será associado a dois SOCs para ser duplamente amostrado, como workaround provido no documento rev0 silicon errata

ADC_setIntPulseGenMode(myAdc, ADC_IntPulseGenMode_Prior); *//Configura o pulso da interrupção ADCINT1 para ocorrer no fim da conversão*

ADC_enableInt(myAdc, ADC_IntNumber_1); *//Habilita a interrupção ADCINT1*

ADC_setIntMode(myAdc, ADC_IntNumber_1, ADC_IntMode_ClearFlag); *//Seleciona ADCINT1 para o modo Clear Flag (uma nova interrupção não pode ser gerada enquanto o flag não for resetado)*

ADC_setIntSrc(myAdc, ADC_IntNumber_1, ADC_IntSrc_EOC2); *//Configura EOC2 (associado a SOC2) para disparar ADCINT1*

ADC_setSocChanNumber (myAdc, ADC_SocNumber_0, ADC_SocChanNumber_A4);
// associa SOC0 no canal ADCINA4

ADC_setSocChanNumber (myAdc, ADC_SocNumber_1, ADC_SocChanNumber_A4);
// associa SOC1 no canal ADCINA4

ADC_setSocChanNumber (myAdc, ADC_SocNumber_2, ADC_SocChanNumber_A2);
// associa SOC2 no canal ADCINA2

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_0, ADC_SocTrigSrc_EPWM1_ADCSOCA);
//seleciona EPWM1A para disparar SOC0 (devido ao round-robin, SOC0 é amostrado antes de SOC1

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_1, ADC_SocTrigSrc_EPWM1_ADCSOCA);
// seleciona EPWM1A para disparar SOC1 (devido ao round-robin, SOC0 é amostrado antes de SOC1

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_2, ADC_SocTrigSrc_EPWM1_ADCSOCA);
// seleciona EPWM1A para disparar SOC2 (devido ao round-robin, SOC1 é amostrado antes de SOC2

ADC_setSocSampleWindow(myAdc, ADC_SocNumber_0,
ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC0 para 7 ciclos de clock, (6 ACQPS + 1)*

ADC_setSocSampleWindow(myAdc, ADC_SocNumber_1,
ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC1 para 7 ciclos de clock, (6 ACQPS + 1)*

ADC_setSocSampleWindow(myAdc, ADC_SocNumber_2,
ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC2 para 7 ciclos de clock, (6 ACQPS + 1)*

// Habilita o clock do módulo PWM1:
CLK_enablePwmClock(myClk, PWM_Number_1);

// Configura ePWM1:
PWM_enableSocAPulse(myPwm); *// Habilita a geração de pulsos para SOCs (grupo A)*

```
    PWM_setSocAPulseSrc(myPwm,  PWM_SocPulseSrc_CounterEqualCmpAIncr);  // Seleciona
    a condição para geração do pulso. Nesse caso, o disparo ocorrerá quando o valor do contador da
    PWM for igual ao valor do comparador A na subida

    PWM_setSocAPeriod(myPwm, PWM_SocPeriod_FirstEvent); // Define que o pulso será gerado
    no primeiro evento

    PWM_setCmpA(myPwm, 0x0080); // Seta o comparador com o valor 0x0080

    PWM_setPeriod(myPwm, 0xFFFF); // Seta o período com o valor 0xFFFF

    PWM_setCounterMode(myPwm,  PWM_CounterMode_Up);  // Define o modo de contagem
    progressivo

    //Habilita o clock do time-base
    CLK_enableTbClockSync(myClk);


    Gpio_setup(); // Chama a função de configuração da GPIO

    LoopCount = 0;

     // Aguarda a interrupção do ADC
    for(;;)
    {
        LoopCount++;
    }
}


__interrupt void adc_isr(void) // Rotina de interrupção
{
    //Devemos descartar o resultado da primeira conversão (SOC0) devido ao workaround de [rev0
    silicon errata]:

    Voltage6 = ADC_readResult(myAdc, ADC_ResultNumber_1); // Recupera o valor da conversão
    do canal ADCINA4, associado à SOC1

    Voltage8 = ADC_readResult(myAdc, ADC_ResultNumber_2); // Recupera o valor da conversão
    do canal ADCINA2, associado à SOC2

    if(Voltage6 < Voltage8) // Compara os valores das tensões
    {
        // Acende o LED1 se a tensão no pino 8 é superior à do pino 6
        GPIO_setHigh(myGpio, GPIO_Number_0);
        GPIO_setLow(myGpio, GPIO_Number_1);
    }
    else
    {
        // Acende o LED0 caso contrário
        GPIO_setLow(myGpio, GPIO_Number_0);
        GPIO_setHigh(myGpio, GPIO_Number_1);
    }

    // Limpa o flag de ADCINT1 para o próximo evento
    ADC_clearIntFlag(myAdc, ADC_IntNumber_1);

    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_10);
}
```

ANEXO 2

```
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo

#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog
#include "f2802x_common/include/timer.h" // inclui biblioteca do cpu-timer


#define period 1000000 // Período em microsegundos

// Protótipo das rotinas de interrupção
__interrupt void adc_isr(void); //Protótipo da rotina de interrupção do ADC
__interrupt void cpu_timer0_isr(void); //Protótipo da rotina de interrupção do cpu-timer0

// Variáveis globais utilizadas nesse exemplo:
uint16_t Voltage6; //Tensão no pino 6
uint16_t Voltage8; //Tensão no pino 8

// Declara os handles globais:
ADC_Handle myAdc;
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm;


void Gpio_setup() // Configura as saídas conectadas aos LEDs no GPIO
{
    // Habilita resistores de pull-up:
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_2, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_3, GPIO_PullUp_Enable);

    // Seta os pinos no modo “propósito geral”:
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_2, GPIO_2_Mode_GeneralPurpose);
    GPIO_setMode(myGpio, GPIO_Number_3, GPIO_3_Mode_GeneralPurpose);

    // Define os pinos como saídas
    GPIO_setDirection(myGpio, GPIO_Number_0, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_1, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_2, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_3, GPIO_Direction_Output);

    // Define o nível lógico inicial dos LEDs:
    GPIO_setHigh(myGpio, GPIO_Number_0);
    GPIO_setHigh(myGpio, GPIO_Number_1);
    GPIO_setHigh(myGpio, GPIO_Number_2);
```

```
    GPIO_setLow(myGpio, GPIO_Number_3);
}

void main(void)
{
    // Declara os handles locais:
    CPU_Handle myCpu;
    PLL_Handle myPll;
    WDOG_Handle myWDog;
    TIMER_Handle myTimer0;

    // Inicializa todos os handles da aplicação:
    myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));
    myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
    myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
    myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
    myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
    myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
    myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
    myPwm = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
    myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));
    myTimer0 = TIMER_init((void *)TIMER0_BASE_ADDR, sizeof(TIMER_Obj));

    // Executa a inicialização básica do sistema:
    WDOG_disable(myWDog);
    CLK_enableAdcClock(myClk); // Habilita o clock do ADC
     (*Device_cal)();

    //Seleciona o oscilador interno 1 para ser o clock:
    CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

    // Configura a PLL para x12 / 2 que resulta em  60Mhz = 10Mhz * 12 / 2:
    PLL_setup(myPll, PLL_Multiplier_12, PLL_DivideSelect_ClkIn_by_2);

    // Desabilita a PIE e todas as interrupções, e limpa os flags:
    PIE_disable(myPie);
    PIE_disableAllInts(myPie);
    CPU_disableGlobalInts(myCpu);
    CPU_clearIntFlags(myCpu);

    // [Usado no modo de configuração "FLASH" do CC] Copia funções da RAM para a RAM:
    #ifdef _FLASH
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
    #endif

    // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
    PIE_setDebugIntVectorTable(myPie);
    PIE_enable(myPie);

    // Configura as rotinas de interrupção na PIE:
    PIE_registerPieIntHandler(myPie,        PIE_GroupNumber_10,        PIE_SubGroupNumber_1,
    (intVec_t)&adc_isr);
    PIE_registerPieIntHandler(myPie,        PIE_GroupNumber_1,        PIE_SubGroupNumber_7,
    (intVec_t)&cpu_timer0_isr);

    // Inicialização básica do ADC:
    ADC_enableBandGap(myAdc); // Habilita o circuito da tensão de referência
    ADC_enableRefBuffers(myAdc); // Habilita o buffer do circuito de referência
    ADC_powerUp(myAdc); // Energiza os circuitos do ADC (ver tópico Power Up Sequence em [4])
```

ADC_enable(myAdc); *// Habilita o ADC (ver tópico Power Up Sequence em [4])*
ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int); *// Seleciona a referência interna*

PIE_enableAdcInt(myPie, ADC_IntNumber_1); *// Habilita ADCINT1 na PIE*

PIE_enableTimer0Int(myPie); *// Habilita timer0 na PIE*

CPU_enableInt(myCpu, CPU_IntNumber_10); *// Habilita CPU Interrupt 10 (grupo 10 da PIE - PIE_GroupNumber_X -> CPU_IntNumber_X; ADC)*

CPU_enableInt(myCpu, CPU_IntNumber_1); *// Habilita CPU Interrupt 1 (timer)*

CPU_enableGlobalInts(myCpu); *// Habilita a interrupção global INTM (ver [5])*

CPU_enableDebugInt(myCpu); *// Habilita a interrupção global do modo de emulação em tempo real DBGM (ver [5])*

// Configuração do ADC
//OBS: O canal ADCINA4 será associado a dois SOCs para ser duplamente amostrado, como workaround provido no documento rev0 silicon errata

ADC_setIntPulseGenMode(myAdc, ADC_IntPulseGenMode_Prior); *//Configura o pulso da interrupção ADCINT1 para ocorrer no fim da conversão*
ADC_enableInt(myAdc, ADC_IntNumber_1); *//Habilita a interrupção ADCINT1*
ADC_setIntMode(myAdc, ADC_IntNumber_1, ADC_IntMode_ClearFlag); *//Seleciona ADCINT1 para o modo Clear Flag (uma nova interrupção não pode ser gerada enquanto o flag não for resetado)*
ADC_setIntSrc(myAdc, ADC_IntNumber_1, ADC_IntSrc_EOC2); *//Configura EOC2 (associado a SOC2) para disparar ADCINT1*

ADC_setSocChanNumber (myAdc, ADC_SocNumber_0, ADC_SocChanNumber_A4); *// associa SOC0 no canal ADCINA4*
ADC_setSocChanNumber (myAdc, ADC_SocNumber_1, ADC_SocChanNumber_A4); *// associa SOC1 no canal ADCINA4*
ADC_setSocChanNumber (myAdc, ADC_SocNumber_2, ADC_SocChanNumber_A2); *// associa SOC2 no canal ADCINA2*

ADC_setSocTrigSrc(myAdc, ADC_SocNumber_0, ADC_SocTrigSrc_CpuTimer_0); *//seleciona cpu_timer0 para disparar SOC0 (devido ao round-robin, SOC0 é amostrado antes de SOC1)*
ADC_setSocTrigSrc(myAdc, ADC_SocNumber_1, ADC_SocTrigSrc_CpuTimer_0); *// seleciona cpu_timer0 para disparar SOC1 (devido ao round-robin, SOC0 é amostrado antes de SOC1)*
ADC_setSocTrigSrc(myAdc, ADC_SocNumber_2, ADC_SocTrigSrc_CpuTimer_0); *// seleciona cpu_timer0 para disparar SOC2 (devido ao round-robin, SOC1 é amostrado antes de SOC2)*

ADC_setSocSampleWindow(myAdc, ADC_SocNumber_0, ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC0 para 7 ciclos de clock, (6 ACQPS + 1)*
ADC_setSocSampleWindow(myAdc, ADC_SocNumber_1, ADC_SocSampleWindow_7_cycles); *//seleciona a sample windows de SOC1 para 7 ciclos de clock, (6 ACQPS + 1)*
ADC_setSocSampleWindow(myAdc, ADC_SocNumber_2, ADC_SocSampleWindow_7_cycles*);* *//seleciona a sample windows de SOC2 para 7 ciclos de clock, (6 ACQPS + 1)*

// Configuração do cpu_timer0

```
    CLK_enableCpuTimerClock(myClk,CLK_CpuTimerNumber_0); // Habilita o Clock de cpu-
    timer0
    TIMER_stop(myTimer0); // Para a contagem do timer
    TIMER_setPeriod(myTimer0, period*60); // Atribui ao timer o período definido no escopo
    TIMER_setPreScaler(myTimer0, 0); // Configura o "prescale" do timer. O clock do timer é dividido
    pelo valor inserido mais uma unidade. Neste caso, será dividido por 1
    TIMER_reload(myTimer0); // Reinicia o timer
    TIMER_setEmulationMode(myTimer0,  TIMER_EmulationMode_StopAfterNextDecrement);  //
    Utilizado na emulação. Define, por exemplo, a ação do timer quando um "breakpoint" é inserido no
    software de desenvolvimento
    TIMER_enableInt(myTimer0); // Habilita a interrupção do timer

    Gpio_setup(); // Chama a função de configuração da GPIO

    TIMER_start(myTimer0); // Inicia a contagem do timer

    // Aguarda uma interrupção
    for(;;);
}


// Rotina de interrupção do timer
__interrupt void cpu_timer0_isr(void)
{
    // Alterna o nível lógico das saídas dos LEDs
    GPIO_toggle(myGpio, GPIO_Number_2);
    GPIO_toggle(myGpio, GPIO_Number_3);

    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_1);
}

// Rotina de interrupção do ADC
__interrupt void adc_isr(void)
{
    //Devemos descartar o resultado da primeira conversão (SOC0) devido ao workaround de [rev0
    silicon errata]:

    Voltage6 = ADC_readResult(myAdc, ADC_ResultNumber_1); // Recupera o valor da conversão
    do canal ADCINA4, associado à SOC1

    Voltage8 = ADC_readResult(myAdc, ADC_ResultNumber_2); // Recupera o valor da conversão
    do canal ADCINA2, associado à SOC2

    if(Voltage6 < Voltage8) // Compara os valores das tensões
    {
        // Acende o LED1 se a tensão no pino 8 é superior à do pino 6
        GPIO_setHigh(myGpio, GPIO_Number_0);
        GPIO_setLow(myGpio, GPIO_Number_1);
    }
    else
    {
        // Acende o LED0 caso contrário
        GPIO_setLow(myGpio, GPIO_Number_0);
        GPIO_setHigh(myGpio, GPIO_Number_1);
    }

    // Limpa o flag de ADCINT1 para o próximo evento
    ADC_clearIntFlag(myAdc, ADC_IntNumber_1);
```

```
    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_10);
}
```

ANEXO 3

```
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo

#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog


#define PWM1_TIMER_TBPRD   0xFFFF // Define o período do timer de ePWM1


void InitEPwmTimer(void); // Declara a rotina de configuração da ePWM
void InitAdc(void); // Declara a rotina de configuração do ADC
void InitGpio(void); // Declara a rotina de configuração do GPIO

__interrupt void adc_isr(void); //Protótipo da rotina de interrupção

uint16_t Voltage; // Variável que armazena o valor da tensão obtido pelo ADC

// Declara os handles globais
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm1;
ADC_Handle myAdc;
CPU_Handle myCpu;


int main(void)
{
    // Declara handles locais
    PLL_Handle myPll;
    WDOG_Handle myWDog;

    // Inicializa os handles
    myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
    myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
    myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
    myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
    myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
    myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
    myPwm1 = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
    myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));
    myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));

    // Executa a inicialização básica do sistema:
    WDOG_disable(myWDog);
    CLK_enableAdcClock(myClk); // Habilita o clock do ADC
    (*Device_cal)();

    //Seleciona o oscilador interno 1 para ser o clock:
```

```
    CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

    // Configura a PLL para x12 / 2 que resulta em  60Mhz = 10Mhz * 12 / 2:
    PLL_setup(myPll, PLL_Multiplier_12, PLL_DivideSelect_ClkIn_by_2);

    // Desabilita a PIE e todas as interrupções, e limpa os flags:
    PIE_disable(myPie);
    PIE_disableAllInts(myPie);
    CPU_disableGlobalInts(myCpu);
    CPU_clearIntFlags(myCpu);

    // [Usado no modo de configuração "FLASH" do CC] Copia funções da RAM para a RAM:
    #ifdef _FLASH
      memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
    #endif

    // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
    PIE_setDebugIntVectorTable(myPie); // *Somente no modo de configuração DEBUG
    PIE_enable(myPie);

    // Configura a rotina de interrupção do ADCINT1 na PIE:
    PIE_registerPieIntHandler(myPie, PIE_GroupNumber_10, PIE_SubGroupNumber_1,
                    (intVec_t)&adc_isr);

    // Chamam as rotinas de configuração do ePWM, ADC e GPIO, respectivamente. Elas serão
    implementadas posteriormente
    InitEPwmTimer();
    InitAdc();
    InitGpio();

    CPU_enableGlobalInts(myCpu); // Habilita a interrupção global INTM (ver ** control interrupt)

    CPU_enableDebugInt(myCpu); // Habilita a interrupção global do modo de emulação em tempo
    real DBGM (ver ** control interrupt)

    for(;;); // Aguarda as interrupções
}


// Rotina de configuração do ePWM
void InitEPwmTimer(void)
{
    CLK_disableTbClockSync(myClk); // Desabilita o clock de todos os TBs

    CLK_enablePwmClock(myClk, PWM_Number_1); // Habilita o clock de ePWM1


    // Configurações do bloco TB
    PWM_setPeriod(myPwm1, PWM1_TIMER_TBPRD); //Define o período com base na definição
    no escopo

    PWM_setCounterMode(myPwm1, PWM_CounterMode_UpDown); // Define contagem prog. e
    regres.

    // Configurações do bloco CC
    PWM_setCmpA(myPwm1, 0); // Define o valor 0 a ser comparado com o contador. Este valor
    será posteriormente substituído pela tensão medida
```

```
    // Configurações do bloco AQ; Notem que as saídas EPWM1A e EPWM1B foram configuradas
    para serem complementares. Poderíamos utilizar o bloco Dead-Band para configurar saídas
    complementares.
    PWM_setActionQual_CntUp_CmpA_PwmA(myPwm1,PWM_ActionQual_Clear); // Leva o
    nível lógico de EPWM1A a 0 quando, na contagem progressiva, o valor de referência for igual ao
    contador

    PWM_setActionQual_CntDown_CmpA_PwmA(myPwm1,PWM_ActionQual_Set); // Leva o
    nível lógico de EPWM1A a 1 quando, na contagem regressiva, o valor de referência for igual ao
    contador

    PWM_setActionQual_CntUp_CmpA_PwmB(myPwm1,PWM_ActionQual_Set); // Leva o nível
    lógico de EPWM1B a 1 quando, na contagem progressiva, o valor de referência for igual ao
    contador

    PWM_setActionQual_CntDown_CmpA_PwmB(myPwm1,PWM_ActionQual_Clear); // Leva o
    nível lógico de EPWM1A a 0 quando, na contagem regressiva, o valor de referência for igual ao
    contador


    PWM_enableSocAPulse(myPwm1); // Habilita a geração de pulsos para SOCs (grupo A)

    PWM_setSocAPulseSrc(myPwm1, PWM_SocPulseSrc_CounterEqualPeriod); // Seleciona a
    condição para geração do pulso. Nesse caso, o disparo ocorrerá quando o valor do contador da
    PWM for igual ao período

    PWM_setSocAPeriod(myPwm1, PWM_SocPeriod_FirstEvent); // Define que o pulso será
    gerado no primeiro evento


    CLK_enableTbClockSync(myClk); // Habilita clock de todos os TB

}


// Configuração do ADC
void InitAdc(void)
{
    // Inicialização básica do ADC:
    ADC_enableBandGap(myAdc); // Habilita o circuito da tensão de referência
    ADC_enableRefBuffers(myAdc); // Habilita o buffer do circuito de referência
    ADC_powerUp(myAdc); // Energiza os circuitos do ADC (ver tópico Power Up Sequence em **)
    ADC_enable(myAdc); // Habilita o ADC (ver tópico Power Up Sequence em **)
    ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int); // Seleciona a referência interna

    PIE_enableAdcInt(myPie, ADC_IntNumber_1); // Habilita ADCINT1 na PIE

    CPU_enableInt(myCpu, CPU_IntNumber_10); // Habilita CPU Interrupt 10 (grupo 10 da PIE -
    PIE_GroupNumber_X -> CPU_IntNumber_X)

    // Configuração do ADC
    //OBS: O canal ADCINA2 será associado a dois SOCs para ser duplamente amostrado, como
    workaround provido no documento [rev0 silicon errata]

    ADC_setIntPulseGenMode(myAdc, ADC_IntPulseGenMode_Prior); //Configura o pulso da
    interrupção ADCINT1 para ocorrer no fim da conversão

    ADC_enableInt(myAdc, ADC_IntNumber_1); //Habilita a interrupção ADCINT1
```

```
    ADC_setIntMode(myAdc, ADC_IntNumber_1, ADC_IntMode_ClearFlag); //Seleciona ADCINT1
    para o modo Clear Flag (uma nova interrupção não pode ser gerada enquanto o flag não for
    resetado)

    ADC_setIntSrc(myAdc, ADC_IntNumber_1, ADC_IntSrc_EOC1); //Configura EOC1 (associado
    a SOC1) para disparar ADCINT1

    ADC_setSocChanNumber (myAdc, ADC_SocNumber_0, ADC_SocChanNumber_A2);
    //associa SOC0 no canal ADCINA2

    ADC_setSocChanNumber (myAdc, ADC_SocNumber_1, ADC_SocChanNumber_A2);
    //associa SOC1 no canal ADCINA2

    ADC_setSocTrigSrc(myAdc, ADC_SocNumber_0, ADC_SocTrigSrc_EPWM1_ADCSOCA);
    //seleciona EPWM1A para disparar SOC0 (devido ao round-robin, SOC0 é amostrado antes de
    SOC1

    ADC_setSocTrigSrc(myAdc, ADC_SocNumber_1, ADC_SocTrigSrc_EPWM1_ADCSOCA);
    //seleciona EPWM1A para disparar SOC1 (devido ao round-robin, SOC0 é amostrado antes de
    SOC1

    ADC_setSocSampleWindow(myAdc, ADC_SocNumber_0,
     ADC_SocSampleWindow_7_cycles); //seleciona a sample windows de SOC0 para 7 ciclos de
    clock, (6 ACQPS + 1)

    ADC_setSocSampleWindow(myAdc, ADC_SocNumber_1,
    ADC_SocSampleWindow_7_cycles); //seleciona a sample windows de SOC1 para 7 ciclos de
    clock, (6 ACQPS + 1)
}


// Configura GPIO
void InitGpio(void)
{
    // Habilita resistores de pull-up
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Enable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Enable);

    // Define os pinos no modo ePWM, que recebe o nível lógico proveniente das saídas dos módulos
    EPWM1A e EPWM1B
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_EPWM1A);
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_EPWM1B);
}


// Rotina de interrupção do ADC
__interrupt void adc_isr(void)
{
    Voltage = ADC_readResult(myAdc, ADC_ResultNumber_1); // Recupera o valor da conversão
    de SOC1, armazenado em ADCRESULT1

    PWM_setCmpA(myPwm1, Voltage<<4); // Define a tensão de referência de CMPA, de forma a
    variar o ciclo de trabalho dass saías EPWM1A e EPWM1B

    // Limpa o flag de ADCINT1 para o próximo evento
    ADC_clearIntFlag(myAdc, ADC_IntNumber_1);

    // Limpa a interrupção no PIE
    PIE_clearInt(myPie, PIE_GroupNumber_10);
```

```
}
```